Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics

Patrick Rempel(✉) and Patrick Mäder

Software Systems Group,
Technische Universität Ilmenau, Ilmenau, Germany
{patrick.rempel,patrick.maeder}@tu-ilmenau.de

Abstract. [**Context and Motivation**] Agile developments follow an iterative procedure with alternating requirements planning and implementation phases boxed into sprints. For every sprint, requirements from the product backlog are selected and appropriate test measures are chosen. [**Question/problem**] Both activities should carefully consider the implementation risk of each requirement. In favor of a successful project, risky requirements should either be deferred or extra test effort should be dedicated on them. Currently, estimating the implementation risk of requirements is mainly based on gut decisions. [**Principal ideas/ results**] The complexity of the graph spanned by dependency and decomposition relations across requirements can be an indicator of implementation risk. In this paper, we propose three metrics to assess and quantify requirement relations. We conducted a study with five industry-scale agile projects and found that the proposed metrics are in fact suitable for estimating implementation risk of requirements. [**Contribution**] Our study of heterogeneous, industrial development projects delivers for the first time evidence that the complexity of a requirements traceability graph is correlated with the error-proneness of the implementing source code. The proposed traceability metrics provide an indicator for requirements' implementation risks. This indicator supports product owners and developers in requirement prioritization and test measure selection.

Keywords: Agile development · Requirements prioritzation · Traceability metrics · Risk estimation

1 Introduction

Agile software development focuses on continuously delivering small but value-added software increments into an integrated baseline, enabling early verification of requirements and architectural assumptions [8]. At the beginning of every increment, requirements are prioritized and the highest-prioritized requirements are chosen for implementation. At the end of every increment, appropriate test measures are applied to verify the requirement implementation. Considering the risk of requirement implementation is beneficial for both activities [4]. Requirements traceability provides support for understanding relations between requirements [21]. Due to our previous work on traceability assessment [26–28], we hypothesized that a systematic assessment of existing trace links can be used to estimate the implementation risk of requirements, and can thus support requirement prioritization and test measure selection in agile projects.

In this paper, we propose three metrics that can be used to systematically assess requirement traceability relations. We conducted an empirical study on five industry-scale agile software projects, each specified by at least 500 requirements artifacts, to investigate whether or not the proposed metrics are appropriate for estimating the implementation risk of individual requirements. The results of our study show that all three metrics are useful to estimate the implementation risk of requirements, and can thus be used to support requirement prioritization and test planning. Furthermore, the results of our study demonstrate that the traceability metrics can also be used as predictors for unseen projects without project-specific training of the predictor.

The remainder of the paper is organized as follows. In Section 2, we discuss why considering the implementation risk of requirements is beneficial for requirement prioritization and test planning in agile projects. In Section 3, we propose three requirements traceability metrics to estimate the implementation risk of requirements in software projects. The empirical study, which we conducted on five industry-scale projects is presented in Section 4, while the data analysis procedure and results are presented in Section 5. Section 6 discusses the results of our study. Potential threats to validity and how we mitigated them are discussed in Section 7. In Section 8, we discuss previous work that is closely related to our study and highlight similarities and differences to our work. We draw conclusions and outline future work in Section 9.

2 Agile Requirements

The idea of agile software development was established through the agile manifesto [2] containing twelve principles. The first two principles of this manifest clearly indicate that requirements in agile developments are treated differently than in plan-driven development processes.

- *Principle 1*: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
- *Principle 2*: "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."

As highlighted by these principles, agile development focuses on continuously delivering small but value-added software increments into an integrated baseline to enable early verification of requirements and architectural assumptions. Thus, requirements need to be prioritized to decide which of them will be implemented with the next increment.

2.1 Requirement Prioritization

Agile approaches have in common that requirements are prioritized based on business value. Higher prioritized requirements are implemented in earlier development increments so that customers can realize the maximum business value [8]. Though, customers and product managers often struggle to perform a justified prioritization, because quantifying the business value is difficult [18,30]. To provide systematic guidance for this task, Cohn [9] identified two important determinants that should be considered when prioritizing agile requirements: the *financial value* of having a feature and the *cost* of developing and maintaining a feature. An important, yet often underestimated aspect are maintenance costs. In a typical life-cycle, 30% of the costs are spent for development and 70% for maintenance [3]. Empirical studies demonstrated that the error-proneness of implemented software is an important driver for maintenance cost [31]. Therefore, estimating the requirement implementation risk by predicting subsequent defects helps to better understand cost, and thus, support requirements prioritization.

2.2 Focusing Test Effort

Beside requirements prioritization, estimating requirement implementation risk is also beneficial for directing testing activities. As critically discussed by Boehm and Turner [4], most projects spend equal time and effort on testing software parts, no matter how risky these parts are. Instead, focusing test efforts on high-risk parts can save downstream maintenance time and effort. Thus, a reliable estimate of requirement implementation risks also supports focusing test efforts.

3 Estimating Implementation Risk of Requirements Through Traceability Metrics

Even though, agile requirements are typically captured in small entities, the *Agile Enterprise Big Picture* [18] illustrates that even simple stories belong to a bigger context and thus have numerous relationships with other requirements. Requirements traceability provides support to make these relationships explicit. Based on the characteristics of the *Agile Enterprise Big Picture* [18] we derived a traceability information model (TIM) [20] for agile requirements management as depicted in Figure 1. This model conceptualizes traceable artifacts and trace links within the context of agile software development. In addition to the decomposition relations, dependency relations may exist between requirement artifacts such as: one artifact conflicts with another artifact, or one artifact supports another artifact. Figure 2 exemplifies a traceability graph containing the four requirement types (epic, feature, story, and task) and the two requirement relation types (decomposition and dependency).

We hypothesize that existing requirement dependency and decomposition relations, materialized as trace links between requirements, can be used to quantify the complexity of relations between requirements in order to estimate the

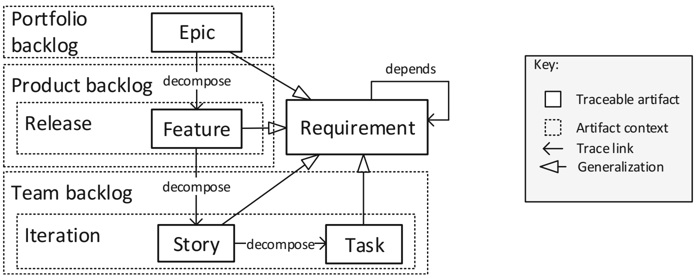

Fig. 1. A traceability information model (TIM) for agile requirements management

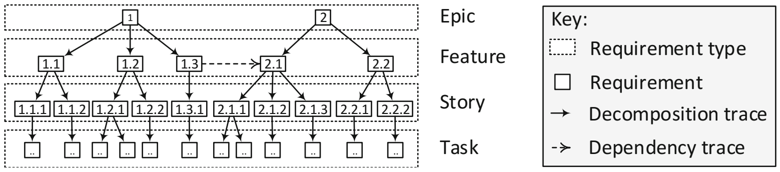

Fig. 2. An exemplary requirements traceability graph including dependencies and decompositions for an agile software development

implementation risk of requirements in agile projects. Therefore, we propose three requirements traceability metrics to assess and quantify the complexity of relationships between requirements. As requirements and trace links span a traceability graph, we aim to characterize the complexity of this graph. In general, a graph consists of vertices and edges, and thus, the complexity is driven by the number of vertices (Section 3.1), the distance between connected vertices (Section 3.2), and the number of edges (Section 3.3).

3.1 Number of Related Requirements (NRR)

Relationships between a requirement and other requirements typically mean that additional requirements and constraints must be considered when implementing that requirement. Thus, with every relation to another requirement, which can be direct or transitive, the complexity of the originated requirement increases. Additionally, a higher number of related requirements implies a higher potential of latent changes when a change request is raised against this requirement.

Definition: The *Number of Related Requirements* (NRR) is the number of requirements that are directly or transitively related to a requirement via decomposition or dependency trace links. A requirement r_j is related to a requirement r_i, if a path of trace links exist from r_i to r_j. RR_i is the set of related requirements of r_i.

$$NRR(r_i) = |RR_i| \tag{1}$$

As exemplified in Figure 3-(A), metric $NRR_{1.3}$ for requirements artifact $r_{1.3}$ is 10 and $NRR_{2.2}$ for requirements artifact $r_{2.2}$ is 12, which means that artifact $r_{2.2}$ is related to more requirements than $r_{1.3}$. The NRR metric computes the same value for all vertices in a connected graph, which could be a limitation. In Figure 3-(A), NRR would be 10 for all requirements connected to $r_{1.3}$. However, new requirements arise continuously in agile projects and thus the traceability graph changes continuously. As the metric is only computed upon the creation or modification of a requirement r_i, NRR is able to discriminate the requirements artifacts of a connected graph.

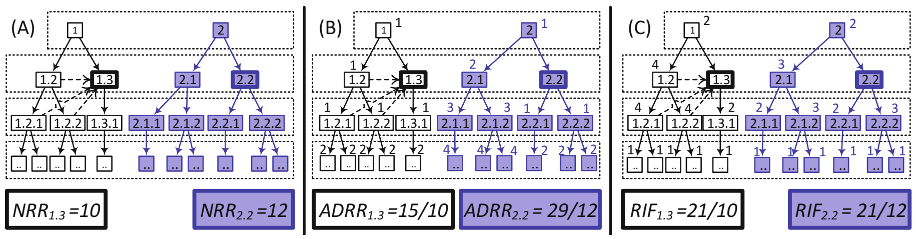

Fig. 3. Examples of the traceability metrics NRR, $ADRR$, and RIF for the requirement artifacts 1.3 and 2.2

3.2 Average Distance to Related Requirements (ADRR)

The distance between two related requirements r_i and r_{ij} indicates how many steps are necessary to traverse the path from r_i to r_{ij}. Longer average distances imply that on average more individual trace links need to be considered by a developer implementing this requirement. The effort for resolving and understanding these relationships increases with longer average distances.

Definition: The *Average Distance to Related Requirements* ($ADRR$) denotes the average number of trace links that need to be resolved to traverse from requirements artifact r_i to any related requirements artifact r_{ij}. The function d_{ij} denotes the distance from r_i to r_{ij}. If alternative paths exit between r_i and r_{ij}, the distance of the shortest path is used for calculation.

$$ADRR(r_i) = \frac{\sum\limits_{r_{ij} \in RR_i} d_{ij}}{|RR_i|} \tag{2}$$

As exemplified in Figure 3-(B), the $ADRR_{1.3}$ for requirement $r_{1.3}$ is $\frac{15}{10}$ and the $ADRR_{2.2}$ for requirement $r_{2.2}$ is $\frac{29}{12}$ suggesting that the average distance to related requirements from $r_{2.2}$ is ~ 0.9 steps longer from $r_{1.3}$.

3.3 Requirement Information Flow (RIF)

The requirement information flow is supposed to determine the coupling of related requirements, which we measure by counting the fan-in and the fan-out of a requirement. An increase in coupling, while assuming a constant number of related requirements, entails an increased number of trace links between the related requirements that must be understood by developers.

Definition: The *Requirement Information Flow* (RIF) of a requirement r_i is the average fan-in and fan-out of any related requirement r_{ij}. The fan-in of a requirement r_{ij} is the number of requirements that are directly connected through an inbound trace link. The fan-out of a requirement r_{ij} is the number of requirements that are directly connected through an outbound trace link. The set of related requirements is denoted as RR_i.

$$RIF(r_i) = \frac{\sum\limits_{r_{ij} \in RR_i} fan\_in(r_{ij}) + fan\_out(r_{ij})}{|RR_i|} \tag{3}$$

As exemplified in Figure 3-(C), metric $RIF_{1.3}$ for requirements artifact $r_{1.3}$ is $\frac{21}{10}$ and metric $RIF_{2.2}$ for requirements artifact $r_{2.2}$ is $\frac{21}{12}$ suggesting that the information flow within the related requirements of $r_{1.3}$ is 0.35 trace links higher than the information flow within related requirements of $r_{2.2}$.

3.4 Research Questions

We hypothesize that the proposed traceability metrics can be used to estimate the requirements implementation risk in order to support the planning activities: requirements prioritization (see Section 2.1) and focusing tests (see Section 2.2). The number of defects at source code level is an accepted metric to quantify the error-proneness of developed software. Since source code is an immediate result of the implementation of requirements, we also consider the number of defects as valid quantification of the requirements error-proneness, and thus, for the requirement implementation risk. Our research questions are as follows:

1. *RQ-1*: Are requirements' NRR, $ADRR$, and RIF metrics associated with the requirements' defects?
2. *RQ-2*: Which, if any, combination of requirements traceability metrics can be used to predict requirements' defects within a project?
3. *RQ-3*: Can predictors, obtained from training projects, also be used to predict the number of defects for an unknown project?

4 Study Design

To investigate our research questions (see Section 3.4), we collected development artifacts and traceability data from five open-source software projects that apply an agile development approach.

4.1 Case Selection

Driven by our research goal to support the requirements prioritization (see Section 2.1) and test selection (see Section 2.2), we defined the following case selection criteria. A case to be included:

- shall apply an agile software development approach (e.g. XP, SCRUM),
- shall provide requirements artifacts at three or more refinement levels,
- shall provide defect artifacts associated to source code and requirements,
- shall provide traceability across requirements, and
- shall be in development for at least five years.

We started our search for potential cases from the list of open source projects[1] that use the ALM tool Jira [17] for requirements management.

4.2 Data Demographics

The five open source projects: CONNECT, Infinispan, jBPM, Weld, and WildFly completely satisfy our case selection criteria and thus, were included in our study. CONNECT is a software project that was initiated by US federal agencies to support their health-related missions. It provides a solution for health information exchange locally and at the national level. Infinispan is a highly available key/value data store and data grid platform. The main purpose is exposing distributed and highly concurrent data structures. The business process management suite jBPM allows modeling and executing business processes. Weld is a reference implementation of the Java standard for dependency injection and contextual lifecycle management: Contexts and Dependency Injection for the Java EE platform. WildFly is a Java application runtime that supports the Java EE 7 standard.

Table 1. Characteristics of the five studied software projects

	Project	**Requirements**					**Trace links**		
		Epic	Feature	Impr.[1]	Task	$\sum$	Dec.[2]	Dep.[3]	$\sum$
A	CONNECT[A]	10	245	290	2,810	**3,355**	3,114	2,538	**5,652**
B	Infinispan[B]	–	850	522	666	**2,038**	372	1,716	**2,088**
C	jBPM[C]	–	1,146	112	1,007	**2,265**	472	1,168	**1,640**
D	Weld[D]	–	242	48	303	**593**	140	620	**760**
E	WildFly[E]	–	682	226	679	**1,587**	798	1,516	**2,314**

[A]www.connectopensource.org, [B]www.infinispan.org, [C]www.jbpm.org, [D]weld.cdi-spec.org, [E]www.wildfly.org, [1]Improvement, [2]Decomposition, [3]Dependency

[1] www.atlassian.com/opensource/overview

Table 1 provides an overview of the requirements artifacts and requirement traceability characteristics of the five studied projects. For all projects, we gained raw data by collecting all relevant project artifacts at the referenced websites created till Aug 16th, 2014. The *Requirements* column shows the number of requirements artifacts per refinement level and in total. The smallest project contains almost 600 requirements artifacts. The *Trace links* column shows the number of relations between requirement artifacts per relation type and in total. The smallest project contains 760 trace links.

4.3 Data Collection Process

Our data collection process consisted of three steps (see Figure 4).

Step 1: Parse Artifacts and Trace Links. All studied projects use the web-based application life-cycle management tool JIRA [17] to manage requirements, defects, and trace links. Every requirement features a unique identifier and trace links between requirements can be navigated forward and backward within the tool. Also, all projects used the source configuration management tool Git [13] to manage source code. We implemented a project artifact collection tool that automatically downloaded and parsed project artifacts and trace links at requirements and source code level. All studied projects were migrated from other requirements management tools to JIRA between 2005 and 2007. To avoid migration influences, we only considered requirements artifacts that were created at least one year after the project was migrated to JIRA.

Step 2: Generate Traceability Graph. Once all relevant artifacts and trace links had been captured by the artifact collection tool (Step 1), a traceability graph could be automatically generated. The generated traceability graph is directed. If a captured trace link is bi-directional, which is the case by default in JIRA, two directed edges are added to the traceability graph.

Step 3: Calculate Traceability Metrics. In the last step, we used the generated traceability graphs to calculate the introduced traceability metrics (see Section 3). Thereby, a data set of traceability metrics was calculated for every issued requirements addition and change. Additionally, we automatically counted the defects that occurred per requirement after this change. All studied projects used an issue tracker system to document defects and their resolution. Project contributors file their discovered defects as issues in this system and thereby support an automated analysis. However, the existence of a defect issue does not necessarily imply the existence of a software defect. Hence, we only considered defects from the issue tracker with the resolution types: *done*, *implemented*, and *fixed*. We excluded all defects with the resolution types: *cannotreproduce*, *communityanswered*, *duplicate*, *goneaway*, *incomplete*, *invalid*, *notaproblem*, *wontfix*, *worksasdesigned*. To correctly count the defects that occurred after a change, we needed to map the defect issues to the affected requirements by extracting two types of relationships. First, our tool analyzed all commit messages within the software configuration management system (SCM) for identifiers of defect issues filed in the issue tracker. Such an identifier means that with this software change the referred defect was addressed. We considered every changed source code file of such a commit to be affected by the defect. Second, our tool analyzed commit messages of the SCM system for identifiers that refer to requirements kept in the issue tracker. Such an identifier means that with this software change the referred requirement was implemented. We considered every changed source file of such a commit to be implementing the requirement. By chaining the extracted relationships: $requirement \xleftarrow{implements} source\text{-}code \xleftarrow{affects} defect$, we could map every defect to one or multiple affected requirements. To make this value comparable for different requirements changes, we always considered the number of defects that occurred within one year after the change. Further, we had to exclude requirement changes from the twelve months prior to our study, since the future defect information were incomplete for these requirements.

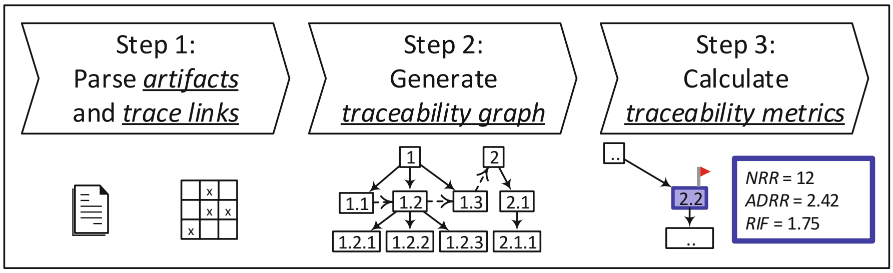

Fig. 4. Overview of the data collection process

5 Data Analysis

5.1 Statistical Model and Regression

As elaborated in Section 3.4, we want to investigate whether or not the proposed requirement traceability metrics NRR, $ADRR$, and RIF are associated with the number of defects occurring in the source code that implements a requirement. For this purpose we apply regression analysis to investigate how the traceability metrics NRR, $ADRR$, and RIF are related to the number of defects per requirement (DEF). That means, NRR, $ADRR$, and RIF are the independent variables and DEF is the dependent variable of our study. Table 2 summarizes the sample size as well as mean and standard deviation for every dependent and independent variable across all five projects.

Table 2. Summary statistics for the studied projects A–E

		DEF		*NRR*		*ADRR*		*RIF*	
	N[1]	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
A	42,278	0.428	1.269	19.489	46.387	3.451	2.677	14.221e+10	9.941e+11
B	22,754	2.61	5.148	10.525	13.583	1.354	1.062	16.874	82.229
C	12,449	0.216	0.676	1.685	1.284	0.442	0.56	2.167	2.615
D	31,024	0.92	2.85	4.863	9.312	1.21	0.471	0.307	0.228
E	59,333	0.538	2.948	2.169	3.524	1.694	0.796	0.597	0.145

[1]N refers to the number of changes calculated as: $\sum_{i=1}^{commits}$ numberOfClasses(i)

Prior to deriving a statistical model explaining the effect of the independent on the dependent variable best, we assessed for every project whether the independent variables correlate with each other by calculating the Pearson correlation coefficient. The purpose was to determine whether or not our proposed traceability metrics are measuring the same characteristics of requirements relationships. If two metrics would strongly correlate with each other for all projects, one metric could be eliminated, because both metrics would measure the same characteristic. Table 3 summarizes the correlation coefficients (*rho*) and their significances (*p-value*). The table shows that no pair of independent variables strongly correlates for all projects. Thus, we considered all three metrics as a potential independent variable influencing the dependent variable.

Table 3. Pearson correlations between independent variables for all projects

	$cor(NRR, ADRR)$		$cor(NRR, RIF)$		$cor(ADRR, RIF)$	
	rho	p-value	rho	p-value	rho	p-value
A	0.621	< 0.001	0.739	< 0.001	0.513	< 0.001
B	0.572	< 0.001	0.249	< 0.001	0.411	< 0.001
C	0.428	< 0.001	0.197	< 0.001	0.239	< 0.001
D	0.636	< 0.001	0.624	< 0.001	0.73	< 0.001
E	0.462	< 0.001	0.95	< 0.001	0.179	< 0.001

To find a statistical model that describes the influence of the three independent variables on the dependent variable, we applied a stepwise model regression in forward direction. We added independent variables stepwise to the statistical model and compared the models of every step with the Akaike Information Criterion (AIC) [7]. With this approach we found that the following equation models the effect of the traceability metrics best for predicting DEF:

$$DEF = \beta_0 + \beta_1(NRR) + \beta_2(ADRR) + \beta_3(RIF) + \epsilon. \quad (4)$$

In Equation 4, the parameters β_1, β_2, and β_3 capture the effect of NRR, $ADRR$, and RIF on DEF. While β_0 is the constant intercept, and ϵ is the error term. First, we estimated the empirical model for Equation 4 with Ordinary Least Square (OLS) regression. However, the Beusch-Pagan test [5] indicated heteroscedasticity for the estimated empirical model, which means that the assumption of homoscedasticity in the regression model was violated. To mitigate this violation, we opted for Weighted Least Square (WLS) regression for fitting the model. The results of the WLS regression are summarized in Table 4.

Table 4. WLS estimates for requirements of all projects

Independent Variable	**Parameter**	**Coefficient**	**Std. error**	**p-value**
Intercept	β_0	0.463 ***	0.009	0.000
NRR	β_1	0.257 ***	0.002	0.000
$ADRR$	β_2	0.352 ***	0.063	0.000
RIF	β_3	-0.068 ***	0.016	0.000

Significance codes for p-values: *** < 0.001, ** < 0.01, * < 0.05

5.2 Predicting Requirement Defects for Unseen Project Data

To assess the generalizability of our statistical model, we applied a leave-one-out cross-validation (LOOCV) strategy. We used the data-sets from four projects as the training sample and the fifth project as the hold-out sample. The regression is fitted on the training sample and applied to the hold-out sample. We repeated this procedure five times, every time with another project as hold-out sample. To evaluate the predictive power on the unseen data, we evaluated how well the predictor can assign a requirement to a risk category. To breakdown the prediction results into usable results, we adapted the traffic light system and distinguished three requirement risk categories:

– *low-risk*: requirement implementation entails 0..2 defects within one year
– *medium-risk*: requirement implementation entails 3..5 defects within one year
– *high-risk*: requirement implementation entails > 5 defects within one year

We considered a requirement risk prediction as correct if the traceability metric predicted the same risk category as we would derive from the actual project defects. Table 5 summarizes the percentage of correctly assigned requirement risk categories per predicted project.

Table 5. Prediction results for unseen project data with LOOCV

	A	B	C	D	E
Correctly predicted risk categories	79.85 %	73.72 %	97.82 %	87.92 %	94.79 %

6 Discussion

The results of the WLS regression for all projects, as shown in Table 4, indicate that the effect of NRR on requirement defects is positive and statistically significant (β_1 is positive and the p-value < 0.001). This result partially answers research question *RQ-1*. It suggest that the more requirements are directly or transitively related to a requirement the higher the defect rate of the requirement's implementation. This relation also implies a higher requirement implementation risk, assuming that all other independent variables remain constant. The WLS regression further indicates that an increase of $ADRR$ is associated with an increase of requirement defects ($\beta_2 = 0.352$), which provides another part to the answer to research question *RQ-1*. It suggests that the longer the average distance of a requirement to directly or transitively related requirements, the higher the requirement's implementation risk. The WLS regression also indicates that an increase of RIF is associated with a decrease of requirements defects (β_3 = -0.068), assuming that all other independent variables are constant.

The fact that none of the three traceability metrics were eliminated during stepwise regression with AIC and that all three metrics are statistically significant for the requirement defects implies that all three metrics have a significant effect on the defect rate in all five projects. This provides the answer to our research question *RQ-2* suggesting that to properly estimate the implementation risk of a requirement, one should consider all three traceability metrics.

The results of the leave-one-out cross-validation (see Table 5) provide an answer to our research question *RQ-3*. For all five cases, our traceability metric predictor assigned requirements to the correct risk-level in unseen projects with at least 73.72 % correctness. For project C and E, the requirement risk predictions are even at a 95 % correctness level. These results suggest that the proposed traceability metrics can be used as an indicator for requirement implementation risks. For practitioners, it provides valuable support with prioritizing requirements as well as for deciding on which part of the implementation testing should be focused. Our proposed metrics can be used to provide recommendations for this tasks as well as to validate existing manual prioritizations.

7 Threats to Validity

A potential threat exists in the preparation and quality of the analyzed project data. Mitigating this threat, we carefully examined the available project artifact types and drew samples manually. These samples confirmed that every requirement artifact is stored as a database record with a unique identifier and trace links can be followed bi-directionally within Jira, from the source to the target requirement and vice versa. To avoid any manual bias during the project data preparation, we fully automated the process of project data collection and analysis. Although the process is fully automated, 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. 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.

Another potential threat exists in the calculation of traceability metrics. The result of a traceability metric directly depends on the completeness and correctness of provided traceability data. In order to identify possible completeness problems, we performed completeness and correctness checks manually where possible. Based on the results of these manual checks, we concluded that all five projects are very mature developments and that the maintained artifacts and trace links are of high quality. Nonetheless, there remains a risk that we may have missed problems, especially incorrect trace links due to the large amount of data and our lack of project-specific domain expertise. Though, the great industrial acceptance and wide dissemination of the software products of all five projects supports our conclusions that all projects are of high maturity.

We analyzed five large scale projects to study the generalizability of our proposed traceability metrics for agile projects. There is a potential threat that one or multiple studied projects could follow an unusual development process which does not represent agile software development properly. To mitigate this threat, we defined project inclusion criteria as described in Section 4.1. Additionally, we manually assessed the project dashboards of all five projects to study the release cycles. We found that all projects follow the Scrum methodology, which is an accepted agile development methodology. We also found that all projects regularly develop and release small software increments in agile Sprints.

Furthermore, the number of defects in agile projects can vary between releases [23]. To mitigate this thread, we chose a rather long future defect observation period of twelve months. Every studied project created at least two releases within that period, meaning that we at least considered two release periods per analyzed project to capture potential post-release defect accumulations. Furthermore, the number of defects may be influenced by additional factors not been investigated within the current study, such as: the number of project members, the number of product end-users, or the experience of the development team. It remains a future exercise to study whether and how such additional factors influence the impact of the requirements complexity on the software defect rate, as measured by the proposed approach.

8 Related Work

Various researchers proposed traceability metrics to characterize traced software artifacts. For example, Pfleger and Bohner [25] proposed software maintenance metrics for traceability graphs. They distinguish vertical and horizontal traceability metrics. While vertical metrics are meant to characterize the developed product, horizontal metrics are meant to characterize the development process. To generically measure the complexity of requirements traceability, Costello et al. [10] proposed the use of statistic linkage metrics. Dick [11] extends the idea of analyzing traceability graphs by introducing trace link semantic, which he calls rich traceability. Main advantage of his approach is the applicability of propositional reasoning to analyze traceability relationships for consistency. Hull and Dick [15] advance the idea of rich traceability graphs and propose further metrics: *breadth* is related to the coverage, *depth* measures the number of layers making it a global metric, *growth* is related to the potential change impact, *balance* measures the distribution of growth factors, *latent change* measures the impact on a change. While all the proposed traceability metrics were meant to measure specific characteristics of the requirements traceability graph, little empirical evidence is available on how and to what extent these metrics support practitioners with activities such as prioritizing requirements or focusing tests.

Researchers also proposed a wide range of metrics to characterize software [12, 29, 32] respectively software design [1, 6, 22]. In contrast to requirements traceability metrics, a wide range of studies provide empirical evidence for software and design metrics on how and to what extent these measures provide support during development. Nagappan et al. [24] as well as Graves et al. [14] study the appropriateness of software component complexity metrics to predict fault density. The implications of software design complexity measures on software defects are empirically investigated by Subramanyam et al. [31].

Although, the principles for measuring the characteristics of a graph are similar, no matter if it represents software (e.g., a software call graph), design (e.g., a component dependency graph), or requirements (e.g., a requirements traceability graph), empirical evidence of the practical benefit of requirement traceability metrics is lacking. As a first step, Mäder and Egyed [19] as well as Jaber et al. [16] conducted controlled experiments to investigate whether or not the existence of traceability data supports developers with software maintenance tasks. To the best of our knowledge, our study is the first to involve large scaled projects providing empirical evidence that requirement traceability metrics can be leveraged to systemically assess and predict the implementation risk of requirements.

9 Conclusions and Future Work

In this paper we focused on estimating the implementation risk of requirements through traceability in agile development projects. Estimating the implementation risk of requirements supports product managers and developers with prioritizing requirements and focusing tests on risky parts of the software.

Therefore, we proposed a set of three traceability metrics to estimate the implementation risk of requirements in this paper. To evaluate the applicability of our proposed metrics, we conducted an empirical study with five large scale agile development projects. The results of our study show that our proposed traceability metrics are suitable to estimate the implementation risk of requirements. We also found empirical evidence that our proposed metrics are generalizable, because the implementation risk of requirements could be estimated reliable in all five projects. Further, our LOOCV experiments show that our proposed traceability metrics can even be applied to unseen project data, making our proposed traceability metrics a potentially valuable tool to support practitioners with prioritizing requirements and making decisions on which part of the software the testing should focus.

Future work will focus on extending and improving the set of requirements traceability metrics to estimate requirement implementation risks. Especially, differences in the requirement structure such as different granularity levels shall be addressed with explicit metrics to improve the generalizability of our metrics. We also plan to extend our study to additional software development projects in order to gain more empirical evidence on the described findings. Finally, projects that do not follow agile development processes shall be included to gain further insights on whether or not the proposed metrics are also beneficial to estimate requirement implementation risks in plan-driven projects. The proposed approach and its evaluation were solely targeting agile development projects. While we hypothesize that the proposed traceability metrics are also applicable for other project types due to their generic nature, additional studies are required to investigate this hypothesis. We also plan to investigate additional potential influence factors such as: the number of project members, the number of product end-users, or the experience of the development team.

Acknowledgments. We are funded by the German Ministry of Education and Research (BMBF) grants: 16V0116, 01IS14026A and the excellence program of the TU Ilmenau.

References

1. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering **28**(1), 4–17 (2002)
2. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al.: The agile manifesto (2001)
3. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001)
4. Boehm, B., Turner, R.: Using risk to balance agile and plan-driven methods. Computer **36**(6), 57–66 (2003)
5. Breusch, T.S., Pagan, A.R.: A simple test for heteroscedasticity and random coefficient variation. Econometrica: Journal of the Econometric Society, 1287–1294 (1979)
6. Briand, L.C., Wüst, J., Daly, J.W., Victor Porter, D.: Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems and Software **51**(3), 245–273 (2000)

7. Burnham, K.P., Anderson, D.R., Huyvaert, K.P.: Aic model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons. Behavioral Ecology and Sociobiology **65**(1), 23–35 (2011)
8. Cao, L., Ramesh, B.: Agile requirements engineering practices: An empirical study. IEEE Software **25**(1), 60–67 (2008)
9. Cohn, M.: Agile estimating and planning. Pearson Education (2006)
10. Costello, R.J., Liu, D.B.: Metrics for requirements engineering. Journal of Systems and Software **29**(1), 39–63 (1995)
11. Dick, J.: Rich traceability. In: Proc. of the 1st Int. Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland, pp. 18–23 (2002)
12. Fenton, N.E., Pfleeger, S.L.: Software metrics: a rigorous and practical approach. PWS Publishing Co. (1998)
13. Git. http://git-scm.com/
14. Graves, T.L., Karr, A.F., Marron, J.S., Siy, H.: Predicting fault incidence using software change history. IEEE TSE **26**(7), 653–661 (2000)
15. Hull, E., Jackson, K., Dick, J.: Requirements engineering. Springer, London (2011)
16. Jaber, K., Sharif, B., Liu, C.: A study on the effect of traceability links in software maintenance. IEEE Access **1**, 726–741 (2013)
17. JIRA. https://www.atlassian.com/software/jira
18. Leffingwell, D.: Agile software requirements: lean requirements practices for teams, programs, and the enterprise. Addison-Wesley Professional (2010)
19. Mäder, P., Egyed, A.: Do developers benefit from requirements traceability when evolving and maintaining a software system? EmpSE, pp. 1–29 (2014)
20. Mäder, P., Gotel, O., Philippow, I.: Getting back to basics: promoting the use of a traceability information model in practice. In: ICSE Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE, pp. 21–25. IEEE (2009)
21. Mäder, P., Gotel, O., Philippow, I.: Motivation matters in the traceability trenches. In: Proc. of the 17th IEEE RE conference, pp. 143–148. IEEE (2009)
22. Marinescu, R.: Measurement and quality in object-oriented design. In: Proc. of the 21st IEEE Int. Conference on Software Maintenance, pp. 701–704. IEEE (2005)
23. Murgia, A., Concas, G., Tonelli, R., Turnu, I.: Empirical study of software quality evolution in open source projects using agile practices. In: Proc. of the 1st International Symposium on Emerging Trends in Software Metrics, p. 11 (2009)
24. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proc. of the 28th Int. Conf. on Software Engineering, pp. 452–461. ACM (2006)
25. Pfleeger, S.L., Bohner, S.A.: A framework for software maintenance metrics. In: Proc. of Software Maintenance conference, pp. 320–327. IEEE (1990)
26. Rempel, P., Mäder, P., Kuschke, T.: An empirical study on project-specific traceability strategies. In: Proceedings of the 21st IEEE International Requirements Engineering Conference (RE), pp. 195–204. IEEE (2013)
27. Rempel, P., Mäder, P., Kuschke, T., Cleland-Huang, J.: Mind the gap: assessing the conformance of software traceability to relevant guidelines. In: Proc. of the 36th International Conference on Software Engineering (ICSE), India (2014)
28. Rempel, P., Mäder, P., Kuschke, T., Philippow, I.: Requirements traceability across organizational boundaries - a survey and taxonomy. In: Doerr, J., Opdahl, A.L. (eds.) REFSQ 2013. LNCS, vol. 7830, pp. 125–140. Springer, Heidelberg (2013)
29. Sedigh-Ali, S., Ghafoor, A., Paul, R.A.: Software engineering metrics for cots-based systems. Computer **34**(5), 44–50 (2001)

30. Sillitti, A., Ceschi, M., Russo, B., Succi, G.: Managing uncertainty in requirements: a survey in documentation-driven and agile companies. In: 11th IEEE International Symposium on Software Metrics, p. 10. IEEE (2005)
31. Subramanyam, R., Krishnan, M.S.: Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Transactions on Software Engineering **29**(4), 297–310 (2003)
32. Washizaki, H., Yamamoto, H., Fukazawa, Y.: A metrics suite for measuring reusability of software components. In: Proceedings of the Ninth International of Software Metrics Symposium, pp. 211–223. IEEE (2003)