

Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments

Alexander Egyed Florian Graf Paul Grünbacher
Systems Engineering and Automation
Johannes Kepler University Linz, Austria
alexander.egyed@jku.at

Abstract—Trace links between requirements and code are essential for many software development and maintenance activities. Despite significant advances in traceability research, creating links remains a human-intensive activity and surprisingly little is known about how humans perform basic tracing tasks. We investigate fundamental research questions regarding the effort and quality of recovering traces between requirements and code. Our paper presents two exploratory experiments conducted with 100 subjects who recovered trace links for two open source software systems in a controlled environment. In the first experiment, subjects recovered trace links between the two systems’ requirements and classes of the implementation. In the second experiment, trace links were established between requirements and individual methods of the implementation. In order to assess the validity of the trace links cast by subjects, key developers of the two software systems participated in our research and provided benchmarks. Our study yields surprising observations: trace capture is surprisingly fast and can be done within minutes even for larger classes; the quality of the captured trace links, while good, does not improve with higher trace effort; and it is not harder though slightly more expensive to recover the trace links for larger, more complex classes.

Keywords—Requirements traceability, traceability effort and quality, exploratory experiments.

I. INTRODUCTION

Requirements traceability is not a new field of research and there is a general consensus among practitioners and researchers that trace links are vital for understanding software systems and for supporting many critical software engineering activities. For instance, trace links between requirements and code identify where requirements are implemented. Such links are required to determine the impact of changes to requirements during maintenance, to perform coverage analyses, or to check the consistency of arbitrary development artifacts [1–3]. Traceability is generally considered most beneficial in long-living software systems [4] when engineers are no longer familiar with the source code [5]. Traceability is nowadays mandated by standards and prescribed in development methods. The existence of trace links is assumed by many existing research approaches [6–8].

However, little is still known regarding the cost-effectiveness of traceability between requirements and code. In domains where system failure implies loss of life or massive economic loss, the question on cost-effectiveness

is secondary. In such domains, trace capture is state of the practice. However, for the vast majority of other systems, the economic benefits are unclear and, as a result, trace capture is rarely done in industrial practice [4]. To understand the cost-effectiveness of traceability, the economic benefits of using traces must outweigh the cost of trace capture and maintenance. *While studies exist that explore the economic benefits of traces [2, 4, 7, 9–11] to the best of our knowledge no studies have explored the cost of trace capture.*

This paper aims to provide this vital missing link. There are many factors that affect the cost of trace capture: the degree of familiarity with a system, the level of automation (for code understanding or trace capture), the availability of documentation, etc. It is clearly impossible to consider all these factors in a study. We thus *focus on understanding the “worst case” of manual trace recovery, i.e., subjects without system familiarity, without system documentation, and without automated support for identifying traces for a given system.* We will show that even in this scenario trace capture is reasonably quick with surprisingly good quality. Our assessment allows researchers to reason about the cost-effectiveness of traceability from a conservative point of view, i.e., if traceability is cost-effective compared to the “worst-case” cost of trace capture then it can be expected that traceability is cost-effective even more so under better circumstances! This work can thus provide a foundation for assessing the economic benefits of traceability for practitioners and researchers alike; for assessing the savings of trace capture automation or the expected cost of their manual overheads; for assessing whether or not a particular use of traces is cost-effective by comparing the savings with cost; and for better understanding the economics of trace capture.

The worst-case situation of recovering trace links between requirements and code is assessed through two exploratory experiments conducted in a controlled environment. The subjects chosen for the experiments were master-level computer science students of Vienna University of Technology and Johannes Kepler University Linz. About half of them had two years or more of industrial experience in software development and the skills and experience of this group is certainly representative of industrial settings. The other half of the subjects had less than two years of industrial experience and the skills of this group are representative of new people

joining companies. None of subjects had a-priori knowledge of the systems used. Our study thus aims at identifying to what degree experienced and inexperienced subjects unfamiliar with the code can recover correct traces. This problem is highly relevant in industry because engineers may understand the bigger picture of the source code and its domain [5, 10] but they often do not understand the purpose of individual classes. As a benchmark for evaluating the subjects' work, we relied on two key developers who wrote the software systems used in the studies and were thus highly familiar with their implementation. Their data allowed us to assess the correctness of the trace links cast by the subjects.

Our study shows that subjects needed on average only 1-2 minutes for recovering the trace links for a class. While the cost of trace recovery increased slightly with code size, we did not find a correlation between code size and the quality of the trace links recovered. Almost all subjects, although unfamiliar with the systems, managed to recover mostly correct traceability (89% on average). However, for the 201,480 traces necessary to completely describe requirements-to-method traces for the first open source system used, 15% incorrectness still amounts to more than 30,000 errors! More experienced subjects did perform slightly better than less experienced subjects but at the expense of higher effort. Our assumption that subjects who investigated a class longer than others would also recover better quality traces could not be confirmed. We found that requirements-to-method traces were 3-6 times more costly to recover than requirements-to-class traces. Surprisingly, however, the correctness of the finer grained method traces was not superior to that of class traces (note that both class and method subjects looked at the exact same code). The recovered traces were more likely correct if the recovery could be done fast. This suggests that subjects quickly had the correct intuition about a class' traceability. While the effort of trace recovery correlated with class size no correlation could be observed between size and correctness. These findings are important for practitioners for better understanding the worst-case cost and quality of trace capture. The findings are also important for researchers to better quantify the cost and benefit of research approaches that rely on the existence of trace links.

II. RELATED WORK

Trace recovery represents a massive re-engineering effort not unlike architecture recovery. To date, the research community has focused largely on automated approaches to recover trace links [12–16]. Despite successes in this field, adequate automation has never been achieved and trace recovery remains a human-intensive activity. Indeed, researchers have pointed out that it is risky to neglect humans in the traceability loop [17]. Nevertheless, *only little is known on how people without system knowledge recover trace links and no data is available on the effort, quality, and complexity of basic trace recovery tasks*. Although trace recovery relies heavily on human expertise to our knowledge so far no experiments have

been conducted to better understand manual trace recovery for large-scale software systems.

Nonetheless, research on traceability has progressed significantly and researchers have been developing automated approaches that go far beyond simple “recording and replaying” of trace links (which is still the level of support in many commercial tools). Approaches exist today that support recovery of different types of trace links such as code and models [18–20], code and documentation [21], architecture and test cases [22], architecture and code [23], or features and code [24]. Researchers have proposed various techniques and heuristics to support the automation of trace recovery. Examples include event-based approaches [25], information retrieval [12, 13], feature location techniques [15], process-oriented approaches [26], scenario-based techniques [14], or rule-based methods [16]. Although advances have been made to automatically recover links, trace acquisition remains a human-intensive activity with high initial cost as reported in case studies on industrial processes and traceability experiences [2, 4, 27–29].

Researchers have also conducted case studies and experiments to determine the effectiveness of traceability approaches. For instance, Hayes *et al.* report on a case study that investigates the effectiveness of information retrieval techniques to create trace links between high-level and low-level requirements [30]. Bianchi *et al.* present an exploratory case study evaluating the relationship between the granularity of the traceability model adopted and the effectiveness of the maintenance process [31]. De Lucia *et al.* describe a controlled experiment [10] on the combined use of traceability links with information retrieval techniques to give hints regarding the similarity of source code elements.

Despite these advances and available heuristics, capturing trace links remains difficult and unreliable. A better understanding of how people recover trace links is essential for researchers aiming at improving their existing techniques. It is also needed by tool developers providing trace recovery features and by practitioners facing the challenges of planning and managing trace recovery activities in industrial practice.

All current automated approaches require some human intervention. Our studies aim at a better assessment on the cost of this human intervention.

III. RESEARCH QUESTIONS

Our research is meant to provide data on effort and quality of trace recovery by using subjects unfamiliar with a system and with no meaningful automation aside of basic reading technologies. We explore five research questions we defined based on the challenges reported in the traceability research literature, research issues discussed at the International Symposium on Grand Challenges in Traceability (GCT'07), as well as our own industrial experiences:

RQ 1. Does code complexity impact trace recovery effort? We explored whether the amount of code engineers have to read and the code's complexity have an impact on trace effort. Our basic assumption was that code of higher complexity

is harder to understand and we expect that *code complexity means higher effort*. We investigated this research question in experiment 1.

RQ 2. Does code complexity impact the correctness of trace links? We investigated to what degree the size and complexity of the source code engineers have to read impacts trace quality. Our expectation is that larger size and higher complexity negatively impact trace link quality. We investigated this research question in experiment 1.

RQ 3. What is the impact of trace granularity on effort? Maximizing the benefits of traceability techniques by applying them in different forms or combinations is a hard research challenge [9]. In earlier research we explored the trade-off between trace granularity and quality [11, 32]. While fine-grained traces (e.g., requirements to methods instead of requirements to classes) increase the possibilities for trace utilization, in many cases their creation is more costly. Based on these earlier results we predict that *fine-grained traces require more effort*. We investigated this issue in experiment 2.

RQ 4. What is the impact of trace granularity on quality? One could assume that fine grained traces are more precise and correct as engineers must investigate each method individually leading to a deeper understanding of a class. Our expectation was that finer granularity positively impacts trace recovery quality. We investigated this research question in experiment 2.

RQ 5. Does the correctness of trace links increase with higher tracing effort? We expect that *more effort implies better quality* which reflects the general belief that people will achieve higher quality if they devote more time. This research question was investigated in experiment 1.

IV. RESEARCH DESIGN

We explored the five research questions on the two open source systems GanttProject (GP) and ReactOS (RO) as shown in Table I.

A. Case Study Systems

GanttProject (GP) <http://ganttproject.biz/> This open source system provides features for project planning and tracking. Users can visualize task dependencies using Gantt charts and compute the start and finish dates of projects. GP supports basic analyses such as critical path computations for identifying tasks delaying the entire project. It also allows the planning of human resources and their degree of involvement in different tasks to optimize staffing. The system was selected as the subjects already had basic skills in project management techniques. Also, GP is easy to use and its key features are explained in existing lectures on project management. The system is quite large, consisting of 41 KLOC Java code distributed in 516 classes and 3689 methods. The part of the system selected for the study consisted of 85 classes and 788 methods.

ReactOS (RO) <http://www.reactos.org/> RO is an open source implementation of the Windows XP OS. RO aims at

Table I
KEY CHARACTERISTICS OF EXPERIMENTS.

	Experiment 1 (Classes)		Experiment 2 (Methods)	
	GP	RO	GP	RO
Subjects	20	20	48	12
Requirements	17	16	17	16
System size	516	245	3,689	3,490
Selected code elements	85	123	788	544
Possible votes of assignment	1,445	1,968	13,396	8,704
Avg. votes cast per subject	748	1,026	1,792	1,282
Votes cast by all subjects	14,966	20,510	86,023	15,379
trace votes	1,068	1,969	3,421	1,104
no trace votes	13,898	18,541	82,602	14,275
Vote redundancy	10.4	10.4	6.4	1.8
% completed (assignment)	51.8%	52.1%	13.4%	14.7%
% completed (entire system)	5.0%	5.0%	2.1%	8.3%

compatibility to XP applications and device drivers. Furthermore, RO provides a graphical user interface that is highly similar to Windows XP (e.g., a start menu, a taskbar, an explorer for performing typical file system operations). Again, the subjects were familiar with the basic functionality of RO due its similarities to the Windows OS. The system is also quite large, consisting of 34 KLOC C++ code distributed in 245 classes (files) and 3490 methods. The part selected for the study consists of 123 classes and 544 methods.

The large size, complexity, and poor documentation made trace recovery a non-trivial exercise. The mnemonic value of variable and method names was quite good which made it fairly intuitive in many cases to guess a trace link. However, these systems hardly contained any comments or other linguistic cues in the code. Subjects investigated parts of the system only and thus were unable even during the experiment to gain deeper knowledge about the system.

B. Experiment Process

We used a series of measures to gauge correctness and completeness. In particular, we used a 3-tiered research design to assess trace link quality: (i) Multiple subjects recovered the traces of any given piece of code to ensure vote redundancy. (ii) We conducted trace recovery independently at two different levels of granularity: In experiment 1, 20 subjects for GP and 20 more subjects of RO investigated how the requirements traced to classes (class traces). In experiment 2, 48 subjects for GP and 12 more subjects RO investigated how the requirements traced to methods (method traces). (iii) We validated the correctness of the trace links cast by the subjects with the help of two key developers of GP and RO. Both had significantly contributed to development and were highly familiar with the code.

The subjects participating in the two experiments were 100 master-level students from Johannes Kepler University Linz and Vienna University of Technology. All subjects were trained in trace recovery and relevant features of GP and RO. As discussed above, our experiments focused on subjects without a-priori system knowledge. For this purpose the cho-

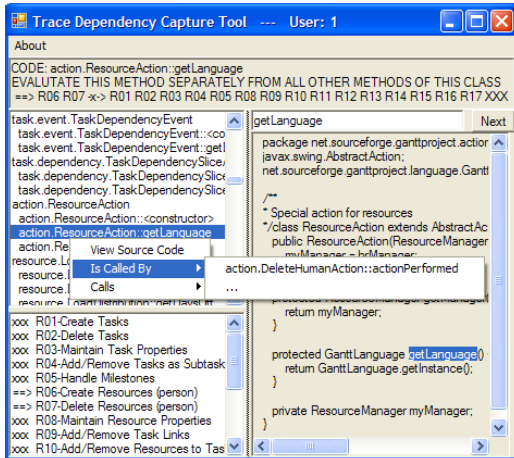


Figure 1. Trace Capture Tool.

sen subjects were an ideal choice since roughly half of them had between 2-10 years of experience and the other half had little to no industrial experience. All subjects were unfamiliar with the source code or implementation details to fit the scope of this study.

We did not know how many classes or methods a subject could recover in a reasonable amount of time as no such benchmark existed. We therefore decided on evaluating only a selected subset of each system and its requirements as shown in Table I. We focused on 17 (GP) / 16 (RO) requirements covering the core functionality of the systems. The requirements were selected randomly – in part together with the developers. Figure 1 shows some of the requirements. The classes were selected based on these requirements. Since the larger majority of classes did not implement any of the selected requirements, irrelevant classes were eliminated with the help of developers and through testing and profiling [14, 33]. While all selected classes were thus relevant to at least one requirement, trace links were still rare. While our selection focused on a subset of requirements and classes, we expected that the average number of requirements per class should remain reasonably small even in larger systems with more requirements. Our experiment setup was thus quite realistic in terms of its low ratio of traces vs. no traces.

In the first exploratory experiment, 20 subjects investigated the trace links among the requirements and the 85 selected Java (GP) classes, another 20 subjects worked on the 123 C++ (RO) classes. In the second experiment the remaining subjects investigated the trace links among those requirements and the 788 (GP) / 544 (RO) methods.

The subjects used a simple *Trace Capture* tool for entering trace links (Figure 1). The tool was purely used for data collection and did not automate trace recovery. For each subject, the tool provided a randomized list of classes or methods (thus presenting a different subset of code to each subject). The tool allowed the subjects to view the source code

of any class or method and provided basic navigation features. Most significantly, the tool revealed the callers and callees of classes and methods. The subjects could navigate and view the source code; and the tool also provided information on parent and child classes. This information was determined through prior static and dynamic analysis of the source code. Initially, all requirement-to-code traces were set as *undefined*. The subjects could change this setting to *trace* or *no trace*. We also advised the subjects to only vote in cases of certainty or otherwise to bypass a vote by leaving it undefined. The tool determined trace recovery effort devoted to different code elements by measuring the time span between selecting the code element, voting on its traceability, and selecting the next one. If a subject returned to a piece of code at a later time then the additional time spent was added.

C. Definitions for Data Analyses

Recovering the trace links for the 85 GP classes and 17 requirements in experiment 1 requires $17 \times 85 = 1,445$ trace/no trace votes for completion. For the 788 GP methods in experiment 2, $17 \times 788 = 13,396$ votes are necessary. In the limited time available, each subject managed to cast a portion of these votes only. For instance, the 20 GP subjects recovering class traces combined cast 14,966 votes with a redundancy of 10.4 votes per requirement and class on average. The remaining 48 subjects cast 86,023 method votes with a redundancy of 6.4 votes per requirement and method. The low redundancy can be explained by the experiment duration of 90 minutes per subject. Even the subset of the system was too large for a subject to cover it entirely. A redundancy of 6 votes per class for 48 subjects (all subjects were given the same classes but in different order) implies that each subject was able to complete on average 13.4% of the required 13,396 votes needed for the 85 classes in the experiment (or 2.1% of the entire 3689 methods). This was intended as we wanted to avoid learning effects of subjects gaining sufficient familiarity with the system during the experiment.

Most votes were *no trace* votes (94%); the remaining 6% were *trace* votes (we ignored votes that subjects left at *undefined*). The overwhelming vote in favor of *no trace* was not surprising given that most classes implement few requirements only. *Trace* links are thus expected to be rare compared to *no traces*. For traceability, however, both the effort of traces and no traces are relevant. During trace recovery, all classes of a system must be investigated and thus the combined effort of all classes is important (whether it traces to a given requirement or it does not). *It would thus be invalid to discard no trace votes from this study.*

The subjects exploring the RO traces also managed to cast a portion of all trace votes of the system in the given time. The 20 subjects who recovered traces to the C++ classes cast 20,510 trace/no trace votes with a redundancy of 10.4 votes per requirement and class. The remaining 12 subject cast 15,379 votes with a redundancy of about two votes on a random subset per class. Even though the RO system received

fewer votes on average, the large number of code elements covered allowed drawing statistically significant conclusions.

Trace correctness in our analyses is measured through the consensus between the trace links captured by the subjects and trace links produced by the developers (benchmark). To compare effort and quality of class and method traces in experiment 2, we combined the method traces to *aggregated classes* using a simple criterion: if at least one of the methods of the class traced to a requirement then the class as a whole was considered to trace to that requirement; otherwise there was no trace.

D. Experiment Data Quality

To control the effort spent, we conducted both exploratory experiments in a controlled environment and supervised the subjects. The sessions were limited to 90 minutes per subject. As discussed above we randomly shuffled the ordering of classes and methods for each subject to ensure that classes and methods had a roughly equal likelihood of being investigated (i.e., this was important since we presumed that subjects would be unable to complete the experiment in the limited time available). Subjects were instructed to emphasize quality over quantity when capturing links. Also, subjects were told to cast *trace* and *no trace* votes in case of confidence only. They could choose to skip votes and leave them *undefined* in case of doubt. However, only 5% of the trace links viewed by the subjects in the tool were skipped and left undefined (all view events were recorded in the tool). Despite the lack of system knowledge, subjects thus seemed to be confident in their votes in most cases.

However, did this confidence also lead to good trace link quality? We compared the data gathered by the subjects to the benchmark provided by the developers: 89% of the GP and RO class and method traces were confirmed by the benchmark. However, the no trace votes were more correct than the trace votes which averaged to 49% (depending on experience). It thus appears to be often easier to rule out a trace link while there is a gray zone where trace recovery is hard. The results show that a very high percentage of trace links discovered by the subjects was confirmed by the benchmark data although the subjects were unfamiliar with the code. Our analyses also confirmed that subjects could not have achieved such a high success rate by random voting.

V. EXPERIMENT 1: CLASS TRACES

In the first experiment, 20 subjects investigated the trace links among 17 requirements and the 85 Java classes of GP. Another 20 subjects investigated the trace links among 16 requirements and the 123 C++ classes of RO. The subjects cast 14,966 trace/no trace votes for the *GP* class traces and 20,510 trace/no trace votes for the *RO* class traces. Since these class votes were roughly evenly distributed among all classes, we ended up with 10.4 votes (for GP and RO) per requirement and class.

A. Code complexity and effort (RQ1)

Research question 1 explored whether the amount of code engineers have to read and the code’s complexity have an impact on trace effort. Trace recovery requires subjects to read source code. It is thus intuitive to assume that larger code size (LOC) and higher code complexity (measured using McCabe’s cyclomatic complexity [34]) increase trace recovery effort. We indeed found a correlation between LOC and effort (Figure 2 shows the data of GP and RO class traces). There is a moderate correlation between size of classes and effort (Figure 2) with Spearman’s rho being 0.508 for GP (p-value <0.00001) and 0.631 for RO (p-value <0.00001).

We split the classes into four categories: 0–30 LOC, 30–100 LOC, 100–300, and >300 LOC (the bucket sizes are evenly distributed in the logarithmic scale used in the figure). Figure 2 shows the median effort for each category. Interestingly, GP effort per class was slightly higher compared to RO effort which implies that the GP classes were harder to assess than RO classes. This might be an effect of the application domain or the understandability of the code. We also measured the impact of McCabe’s cyclomatic complexity on effort and observed the same effect with Spearman’s rho being 0.387 for GP (p-value 0.00013). Overall, the results show that larger classes are more expensive to trace than smaller classes but at decreasing marginal costs (a 10-fold LOC increase corresponds roughly to a 2-fold effort increase).

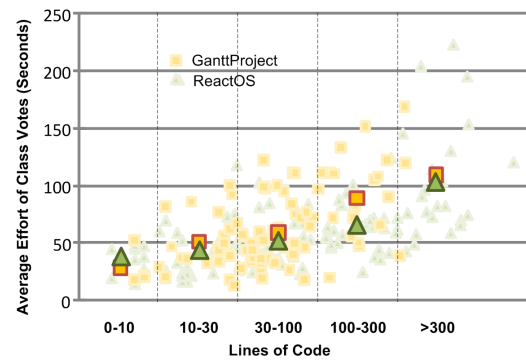


Figure 2. Moderate correlation between code size and effort for GP/RO.

B. Code complexity and quality (RQ2)

It is also intuitive to believe that trace link quality decreases with higher code complexity – the larger and the more complex a class, the more likely errors should occur. We assessed the impact of code size and McCabe’s cyclomatic complexity on trace link quality (measured by the number of conflicts with developers). Given the above observations, we expected trace recovery to be easier for smaller and less complex classes.

Figure 3 shows however that there is no correlation between the size of classes and the average number of conflicts for GP (rho 0.063). There is also no correlation between McCabe’s

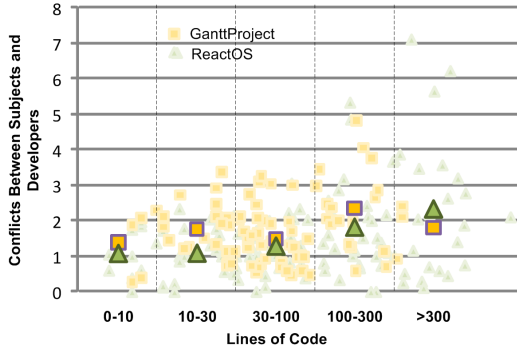


Figure 3. Code size is not correlated with trace quality.

cyclomatic complexity and the number of conflicts for GP with Spearman’s rho being -0.01 for GP (p-value 0.5379). There is however a weak correlation between the size of classes and the average number of conflicts for RO (rho 0.290, p-value 0.00057). While LOC and McCabe do not consider more complex features of source code (inheritance, cohesion and coupling), this observation nonetheless suggests that certain syntactic features of source code do not much affect trace link quality. Only the meaning and usage seems relevant.

VI. EXPERIMENT 2: METHOD TRACES

In experiment 2 we gathered data on trace recovery between requirements and methods. The purpose was to better understand the impact of trace granularity on trace effort by comparing data with results from experiment 1 (RQ3). We also aimed at deeper analyses regarding trace granularity and quality (RQ4). In experiment 2, 48 subjects investigated the trace links among the GP requirements and the 788 GP methods. Another 12 subjects investigated the trace links among the RO requirements and the 544 RO methods. Considering GP we required more subjects to achieve the same code coverage as in experiment 1, mainly because assessing trace links at a finer level of granularity increases the number of code elements (788 methods versus 85 classes). The 48 subjects cast 86,023 trace/no trace votes for method traces. For the 788 methods, the 48 subjects produced a redundancy of roughly 6 votes per requirement and method. These votes were aggregated to make them comparable to the class traces. In addition, for RO, we used 12 subjects to identify trace links between the 544 RO methods and the 16 requirements. They cast 15,379 trace/no trace votes with a coverage of roughly 2 votes per requirement and method. This lower redundancy is compensated by a larger number of methods.

A. Trace granularity and Effort (RQ3)

Regarding this research question, we tried to understand the impact of trace granularity on effort. We expected that fine-grained trace recovery requires more effort since methods must be investigated individually.

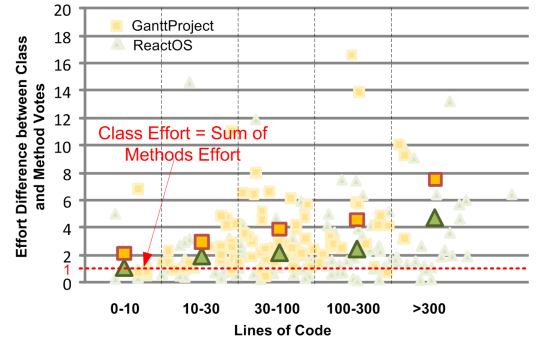


Figure 4. Method votes required higher effort than class votes. The ratio is shown for GP and RO and increases for larger classes. The dotted line describes the situation of equal effort between class and method votes.

Figure 4 shows a ratio expressing the relation of the effort for recovering method traces vs. recovering class traces. If the sum of the individual method efforts of a class exceeds the class effort as a whole then it is above the nominal line (red, dashed line), otherwise it is below (equal effort yields a ratio of 1). There is a moderate correlation between the size and the method-to-class effort ratio (Figure 4) for both GP (rho 0.288, p-value 0.0038) and RO (rho 0.366, p-value 0.00001). There is also a moderate correlation between McCabe’s cyclomatic complexity and the number of conflicts for GP (rho 0.272, p-value 0.00578). This confirms our expectation. However, it is surprising that the increase for class traces was not as strong as the increase for method traces. This observation suggests that class trace recovery is easier because understanding the purpose of a class as a whole is easier than understanding the purposes of each of its methods combined. It also appears that subjects do not need to investigate all methods of a class to establish trace links at the granularity of classes, which might explain the significantly weaker increase of class trace link effort compared to method trace link effort.

B. Trace Granularity and Quality (RQ4)

We expected that fine-grained trace recovery requires more effort but at the same time produces trace links of better quality due to the higher work precision needed. We thus assessed the impact of code size and complexity on trace link quality (measured by the number of errors compared to developer benchmark). Given the above observations, we already showed that trace recovery was not easier for smaller and less complex trace links at the granularity of classes.

However, there were some disagreements between classes and method votes. The above discussion merely shows that code size and complexity do not account for these differences. Of the 94 (GP) / 45 (RO) class traces and 103 (GP) / 38 (RO) aggregated method traces, class traces and method traces agreed in 41 (GP) / 25 (RO) cases. 46 (GP) / 17 (RO) class traces were not found at the granularity of methods while 55 (GP) / 10 (RO) aggregated method traces were not found at the granularity of classes. For better understanding agree-

Table II
COMPARISON OF CLASS AND METHOD TRACE LINKS WITH THE DEVELOPER BENCHMARK.

Class Group	Methods Group	Developer Benchmark	GP	RO
trace	trace	confirm	41	25
		reject	7	3
no trace	no trace	confirm	1,036	396
		reject	56	14
no trace	trace	with class	27	5
		with method	28	5
trace	no trace	with class	23	15
		with method	23	2

ments and disagreements between class traces and method traces, we compared the trace votes of the subjects with the benchmark of the two developers. We expected that class traces would be of lower quality and consequently most of these missing or extra traces should be the result of incorrect class traces (and not incorrect method traces).

Table II summarizes our findings for both systems. We see, for example, in row 4 that of the traces found by the class group but not found at the granularity of methods, the developer agreed with the class group in 23 cases and with the methods group in 23 cases. In the case of the traces found by the method group but not at the granularity of classes (row 3), the assessment group agreed with the class group in 27 cases and the method group in 28 cases. The data gathered on RO was very similar (see right column). Indeed, our expectation was wrong and subjects working on methods traces did not produce better quality traces than subjects working on class traces. We thus conclude that method traces are not of better quality than class traces – despite 3-6 times higher effort.

VII. TRACE EFFORT AND QUALITY (RQ5)

Research question 5 investigates how trace recovery effort is correlated with trace correctness. It is a general truism that effort and quality are positively correlated. But, as was already revealed in the comparison of class vs. method traces, finer-grained traces require 3-6 times more effort to produce without a correlation between finer and coarser-grained traces and trace quality. This suggests that from the perspective of trace quality there is no benefit in increasing effort by asking subjects to investigate classes in more detail. One might argue that recovering method traces and class traces is somewhat different. We thus also investigated the role of effort on class traces only. We investigated the effort of different subjects for all classes and surprisingly observed that the subjects who spent more time on a class were also more likely to recover incorrect trace links.

To illustrate this, we divided all classes into four equally sized buckets according to the effort the subjects spent to cast their votes. Figure 5 shows boxplots of the four buckets based on the average number of conflicts of the trace votes with the benchmark data. Surprisingly, the more effort subjects spent on a given class the more likely their votes were in conflict

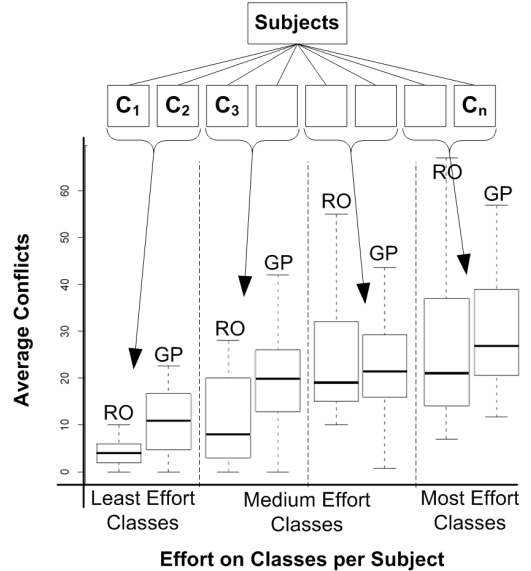


Figure 5. More effort spent on a class resulted in more conflicts. The right bucket contains conflicts of the fastest subjects for each class.

with the benchmark. The rightmost bucket, for example, contains the conflicts of classes where the subjects spent the most effort. This bucket with the slowest classes contains more conflicts than the bucket with the fastest classes. Trace links cast on RO produced fewer conflicts than trace votes casted on GP. Nevertheless, in both cases a higher effort spent on a class resulted in poorer quality.

This observation is not immediately intuitive. Our explanation is that easy trace links required little effort and could be produced with high correctness. However, hard trace links required more effort and were of lower quality due to their complexity *despite the extra effort*. Note that subjects decided freely how much time they would spend on a class. This data thus does not yield insights into whether the quality would improve if the subjects were forced to spend more time. However, it should be noted that this aspect was investigated with the class vs. method traces where *we forced subjects to investigate all methods individually which yielded a higher effort cost but no quality improvements*.

Regarding RQ5 we conclude that more effort does not yield better trace quality – neither by changing granularity nor through longer evaluation times. Trace recovery is fast and accurate when the assigned classes are easy to comprehend. Otherwise, it is hard and inaccurate despite the extra effort invested. Asking subjects to simply spend more time does not obviously benefit trace quality. This suggests that trace quality should be improved by means other than effort – perhaps automation or code familiarity.

VIII. THREATS TO VALIDITY

As any empirical study, our exploratory experiments exhibit a number of threats to validity [35].

A threat to *construct validity* – are we measuring what we mean to measure – is the potential bias caused by the systems selected for the experiment meaning that our experiment may underrepresent the construct. However, we used two fairly large software systems developed by multiple people and with different implementation languages (Java and C++). Furthermore, both systems have gone through multiple revision cycles (thus exhibiting aging effects). We therefore believe that they represent typical systems found in industry that are no longer understood by their developers.

A threat to *internal validity* – are the results due solely to our manipulations – is selection, in particular the assignments of code elements to particular subjects. We used randomization and changed the ordering of code elements to avoid systematic bias from selection. A second threat to internal validity is process conformance. However, the trace capture tool and the supervision enabled us to easily ensure process conformance. Data consistency was ensured during the experiment due to tool support. Supervisors collected the trace and effort log data immediately after each step to avoid manipulation. Additionally, we optimized the measurement of effort by fine-grained tracking of user actions.

We are also able to rule out random subject voting as a significant threat to validity. The subjects cast 50% correct trace votes and 95% correct no trace votes. Both trace and no trace votes must be considered together. With random guessing, one would cast 50% correct traces (as the students did) but then also only cast 50% correct no traces (while the students cast 95% correct no traces). By simply voting *no trace* in all cases, one would cast 95% correct no traces (as the students did) but 0% correct trace votes (while the students cast 50% correct trace votes). The students however found significant numbers of correct traces and no traces. The results in this study are thus the result of ability – not luck.

Moreover, one might expect a startup phase where trace recovery is slow and a fatigue effect setting in after prolonged trace recovery. E.g., other researchers have suggested that trace recovery should be done incrementally, in short but frequent sessions [36]. We found that subjects worked near optimal after only 20 minutes. Our findings thus were not biased much by the experiment setup (data excluded for brevity).

Regarding *conclusion validity* the high number of classes/methods investigated and the large number of subjects allowed us to demonstrate statistical significance of the results in both experiments.

With respect to *external validity* – can we generalize the results – we took two real-world, large systems representing realistic application contexts. The size of the systems is similar to related experiments [10] but not particularly high compared to documents in industrial settings. For instance, we took 85 classes representative of GP as whole. The question is whether these classes are representative of Java classes in general? The selection of the 85 from 450 classes was

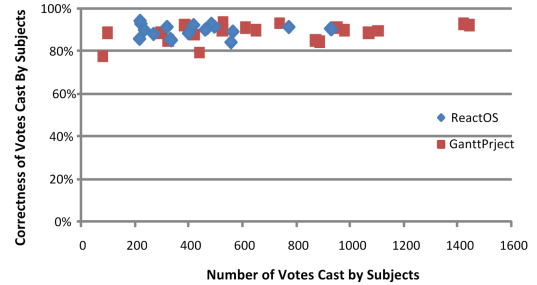


Figure 6. Performance of subjects.

based on static and dynamic analyses of 17 randomly selected requirements so we can assume a reasonably random distribution of hard and easy classes.

In many industrial settings people have no intimate system knowledge during trace recovery. The experiments thus investigated whether subjects unfamiliar with the source code can successfully perform trace tasks. The subjects were students participating in classes on requirements engineering. It has been pointed out that students may not be representative of real developers. However, for the scope of our study the selection of students as subjects does not represent a threat to validity as they are representative of the group of people joining companies and needing to familiarize themselves with source code. The students certainly had the necessary technical skills to perform basic trace recovery tasks as the quality of their data shows. Also, Höst *et al.* observe no significant differences between students and professionals for small tasks of judgment [37], a condition that is met in our case. Moreover, to assess validity of the trace links cast, key developers of the open source systems developed a benchmark. Trace links of subjects and developers overlap – another suggestion that the subjects performed well. The high correctness of the trace links (compared to benchmark data) shows that students certainly had the technical necessary skills to perform the trace recovery task.

Figure 6 shows that subjects performed very well – recovering between 80-95% correct trace/no trace votes. However, as was discussed earlier, this high quality is slightly misleading as the number of no trace votes dwarves the trace votes which were only 49% correct for RO and merely 37% correct for GP (both averages across all subjects). However, this quality difference is not surprising and the overall 80-95% quality is quite outstanding.

Finally, many subjects are already professional software developers with several years of industrial experience. A detailed analysis of the subjects is shown in Figure 7. 55% of subjects had less than 2 years of industrial experience, 24% of subjects had 2-4 years of industrial experience, and 21% of subjects had more than 4 years of industrial experience.

The lower red line in the left figure shows that experienced subjects outperformed novices with respect to trace correctness, however, at the expense of trace effort (green line in

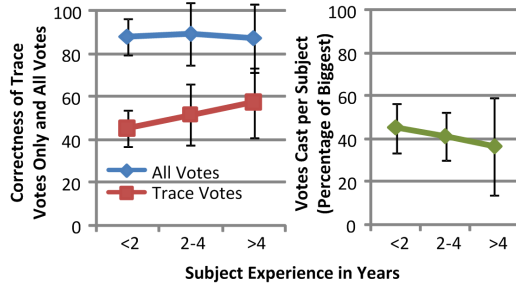


Figure 7. Experienced subjects perform better but need longer.

the right figure). We can report positive correlation between the experience of subjects and the correctness of trace links was significant (ρ 0.313, p -value 0.0217). There is however no correlation between the experience of subjects and the correctness of all trace links. Experience thus seems to matter little during trace recovery: another indication that students are well suited as subjects.

IX. SUMMARY AND DISCUSSION

We presented the results of two exploratory experiments on recovering trace links between requirements and code. We believe that our exploratory studies are valuable as they succeeded in confirming and dismissing some existing beliefs and in providing a foundation for assessing the cost of manual trace recovery under "worst case" assumptions. The latter is important for assessing the cost-effectiveness of any technology that relies on traceability.

We intentionally selected subjects that were unfamiliar with the systems, supplied no automated support for trace recovery, and provided no documentation for the task at hand – with the intent of creating a "worst case" environment. During the course of the experiments, individual subjects investigated a small set of classes only which did not allow them to gain a reasonable understanding of the system. Documentation of the source code was non-existent with the exception of very few comments. This is largely consistent with industrial settings where the original developers of a system are either no longer available or are no longer intimately familiar with the very details of the system [5, 10]. Despite these constraints trace quality was surprisingly high (with *no trace* links easier to determine correctly compared to *trace* links).

Regarding research question RQ1, the data indicates that increased code complexity was associated with an increased trace recovery effort. This data is not a contradiction to our observations in RQ5. Larger classes do need more time to recover than smaller classes. *However, for any given class, more effort does not mean better quality.* Regarding research question RQ2 the results of our analyses reveal that there is no correlation between the code size/complexity and the quality of the trace links. This suggests that quality of trace recovery is not determined by syntactic facts but rather semantic facts such as the meaning of identifiers or the context of code frag-

ments. In future work we will analyze the navigation behavior of subjects to find out whether more than local knowledge is required in more complex cases. Regarding research question RQ3 our experiment shows that tracing requirements to methods required 3-6 times more effort than tracing requirements to classes. However, traces at the granularity of methods have no advantage over traces on granularity of classes in terms of trace quality (RQ4). We got surprising results regarding RQ5 as a higher tracing effort does not imply better quality. Data indicates that trace link recovery is either fast and accurate or slow and inaccurate.

X. CONCLUSIONS

Automation is critical to support trace recovery but still in its infancy. Existing commercial tools help recording and managing traces but they don't help recover them. We hope that the knowledge gained in this study can help researchers and tool builders to automate trace recovery. Our work was also motivated by the fact that there exists no large system with known trace links for researching the problem of trace recovery. Our data provides a meaningful benchmark that can be used and further refined by other researchers in the community who need to assess the effectiveness and efficiency of automated traceability approaches.

For practitioners, our study reveals interesting facts about the cost and quality of trace capture in the "worst case" scenario of unfamiliar subjects without tool support: Trace recovery requires on average 1-2 minutes depending on code size. The quality of trace recovery favors no trace votes over trace votes. It appears to be much easier to correctly eliminate a class from tracing to a requirement than including it. Yet, we could not find a correlation between code size and quality. Trace recovery of method traces costs between 3-6 times as much as class traces. Most interesting, however, the quality of method traces is no better than that of class traces despite the higher effort.

Future work will investigate more precisely the relationship between trace cost/quality to code structure and complexity (e.g., coupling and cohesion). Furthermore, we plan on investigating separately the cost and quality implications for trace and no trace votes.

REFERENCES

- [1] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. RE Conf.*, pp. 94–101, 1994.
- [2] M. Lindvall and K. Sandahl. Practical implications of traceability. *Sw. Pract. Exper.*, 26(10):1161–1180, 1996.
- [3] B. Ramesh and M. Jarke. Toward reference models of requirements traceability. *IEEE TSE*, 27(1):58–93, 2001.
- [4] B. Ramesh, L. C. Stubbs, and M. Edwards. Lessons learned from implementing requirements traceability. *Crosstalk – J of Defense Sw Eng*, 8(4):11–15, 1995.

- [5] D. L. Parnas. Software aging. In *Proc. ICSE*, pp. 279–287. 1994.
- [6] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems J*, 45(3):515–526, 2006.
- [7] M. Deng, R. E. K. Stirewalt, and B. H. C. Cheng. Retrieval by construction: a traceability technique to support verification and validation of UML formalizations. *IJSEKE*, 15(5):837–872, 2005.
- [8] L. C. Briand, Y. Labiche, and T. Yue. Automated traceability analysis for UML model refinements. *IST J*, 51(2):512–527, 2009.
- [9] J. Cleland-Huang, G. Zement, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *12th IEEE Int'l Conf. on RE*, pp. 230–239. 2004.
- [10] A. de Lucia, R. Oliveto, F. Zurolo, and M. D. Penta. Improving comprehensibility of source code via traceability information: a controlled experiment. In *14th Int'l Conf. on Program Comprehension (ICPC 2006)*, pp. 317–326. 2006.
- [11] A. Egyed, S. Biffi, M. Heindl, and P. Grünbacher. Determining the cost-quality trade-off for automated software traceability. In *Proc. ASE Conf.*, pp. 360–363. 2005.
- [12] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settini, and E. Romanova. Best practices for automated traceability. *IEEE Computer*, 40(6):27–35, 2007.
- [13] C. Duan and J. Cleland-Huang. Clustering support for automated tracing. In *Proc. ASE Conf.*, pp. 244–253. 2007.
- [14] A. Egyed. A scenario-driven approach to traceability. In *Proc. ICSE*, pp. 123–132. 2001.
- [15] R. Koschke and J. Quante. On dynamic feature location. In *Proc. ASE Conf.*, pp. 86–95. 2005.
- [16] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause. Rule-based generation of requirements traceability relations. *JSS*, 72(2):105–127, 2004.
- [17] J. H. Hayes and A. Dekhtyar. Humans in the traceability loop: can't live with 'em, can't live without 'em. In *Proc. 3rd Int'l WS on Traceability in emerging forms of sw. eng.*, pp. 20–23. 2005.
- [18] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: selecting the basic linkage properties. *Sci. Comput. Program.*, 40(2–3):213–234, 2001.
- [19] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proc. ASE Conf.*, pp. 163–171. 2002.
- [20] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE*, 27(4):364–380, 2001.
- [21] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. ICSE*, pp. 125–137. 2003.
- [22] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE TSE*, 30(3):160–171, 2004.
- [23] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner. Continuous and automated evolution of architecture-to-implementation traceability links. *ASE J*, 15(1):75–107, 2008.
- [24] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Proc. ASE Conf.*, pp. 254–263. 2007.
- [25] J. Cleland-Huang, C. K. Chang, and M. J. Christensen. Event-based traceability for managing evolutionary change. *IEEE TSE*, 29(9):796–810, 2003.
- [26] K. Pohl. PRO-ART: Enabling requirements pre-traceability. In *Proc. ICSE*, pp. 76–85. 1996.
- [27] H. U. Asuncion, F. François, and R. N. Taylor. An end-to-end industrial software traceability tool. In *Proc. ESEC/FSE*, pp. 115–124. 2007.
- [28] O. Gotel and A. Finkelstein. Extended requirements traceability: Results of an industrial case study. In *Proc. RE Conf.*, p. 169ff. 1997.
- [29] C. Neumüller and P. Grünbacher. Automating software traceability in very small companies: A case study and lessons learned. In *Proc. ASE Conf.*, pp. 145–156. 2006.
- [30] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, and S. Howard. Helping analysts trace requirements: An objective look. In *Proc. RE Conf.*, pp. 249–259. 2004.
- [31] A. Bianchi, G. Visaggio, and A. R. Fasolino. An exploratory case study of the maintenance effectiveness of traceability models. In *8th Int'l WS on Program Comprehension*, p. 149ff. 2000.
- [32] A. Egyed, P. Grünbacher, M. Heindl, and S. Biffi. Value-based requirements traceability: Lessons learned. In *Proc. RE Conf.*, pp. 115–118. 2007.
- [33] A. Egyed, G. Binder, and P. Grünbacher. Strada: A tool for scenario-based feature-to-code trace detection and analysis. In *Proc. ICSE Conf.*, pp. 41–42. 2007.
- [34] T. J. McCabe. A complexity measure. *IEEE TSE*, 2(4):308–320, 1976.
- [35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
- [36] A. de Lucia, R. Oliveto, and G. Tortora. IR-based traceability recovery processes: An empirical comparison of "one-shot" and incremental processes. In *Proc. ASE Conf.*, pp. 39–48. 2008.
- [37] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.