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

Alexander Egyed Florian Graf Paul Grünbacher
Institute for Systems Engineering and Automation
Johannes Kepler Universität
Altenbergerstr. 69, 4040 Linz, Austria
e-mail: {alexander.egyed | florian.graf | paul.gruenbacher}@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 (half with industrial experience, the other half without) who recovered trace links for two open source software systems in a controlled environment and cast over 125.000 votes. In the first experiment, subjects recovered trace links between the two systems' requirements and implementation classes. In the second experiment trace links were established between requirements and implementation methods. 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; it is not harder though slightly more expensive to recover the trace links for larger, more complex classes; and, trace capture should be performed by multiple engineers because "hard-to-do" traces differed per subject and were not uniform to certain parts of code. These findings open interesting possibilities for future research.**

Keywords – Requirements, traceability, cost, understanding.

I. INTRODUCTION

Trace links between requirements and code identify where requirements are implemented. They are important for many success-critical development and maintenance activities. 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 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 [19, 24, 33]. Traceability is generally considered most beneficial in long-living software systems [34] when engineers are no longer familiar with the source code [31]. Traceability is nowadays mandated by standards and prescribed in development methods. The existence of trace links is assumed by many existing research approaches [1, 11, 38].

However, little is known on 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 [34]. 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 [7,10,11,14,24,34] to the best of our knowledge no studies have explored the cost of trace capture and maintenance.*

This paper aims to provide this vital missing link. There are many factors that affect the cost of trace capture: the degree of familiarly 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, i.e., fully manual trace recovery by subjects without system familiarity, without automated support in identifying traces, and without additional documentation about the system*. We will show that even in this extreme scenario, trace capture is reasonably quick with surprisingly good quality. This worst-case assessment allows researchers to reason about the cost effectiveness of traceability from a conservative point of view – that is, if traceability is cost effective compared to the worst-case cost of trace capture then it can be argued that traceability is cost effective under all circumstances (and more so under better circumstances)! This work thus provides 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*
- *for simply better understanding the economics of trace capture which are not known*

The worst-case assessment of trace capture is done through two exploratory experiments [36] conducted in a controlled environment on recovering trace links between requirements

and code. 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 2 years or more of industrial experience in software development and the skills and experience of those subjects is certainly representative of industrial settings. The other half of the subjects had less than two years of industrial experience and their skills are representative of new people joining companies. Both groups of subjects had no 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 still recover correct traces. *This problem is highly relevant in industry because engineers may understand the bigger picture of the source code and its domain [10, 31] but they often do not understand the purpose of individual classes.* Since our study aims at identifying the worst case cost of trace capture, the choice of students as subjects is ideal: these students are nearly finished in their studies and represent soon-to-be software engineers in companies. As a benchmark for evaluating the subjects' work, we relied on two key developers who wrote the software systems in the studies and thus were highly familiar with its implementation. Their data allowed us to assess the correctness of the trace links cast by the subjects.

Our study shows that subjects needed in average only 1-2 minutes for recovering the trace link for a class. While the cost of trace recovery increases slightly with code size, we did not find a strong 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 (80-90%). 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 30.000 errors! Surprising was that subjects were performing at peak efficiency after only 20 minutes after the start of the experiments, with a noticeable exhaustion after 90 minutes. Trace recovery should thus be done incrementally. More experienced subjects did perform slightly better than less experienced subjects but at the expense of higher effort. But our assumption, that subjects who investigated a class longer than others would also recover better quality traces was wrong: we found that the more detailed requirement-to-method traces were 3-6 times more costly to recover than requirements-to-class traces. Surprisingly, however, the correctness of finer grained method traces was not superior to class traces. Trace recovery was more likely correct if the recovery was fast. This suggests that subjects either quickly had the correct intuition about a class' traceability or they did not. While the effort of trace recovery correlated strongly with class size and complexity, quality had only a weak correlation with size/complexity.

Practical Implications: The 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/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 [5, 12, 13, 23, 35]. 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 [20]. 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 [2, 16, 28], code and documentation [25], architecture and test cases [27], architecture and code [29], or features and code [8]. Researchers have proposed various techniques and heuristics to support the automation of trace recovery. Examples include event-based approaches [6], information retrieval [5, 12], feature location techniques [23], process-oriented approaches [32], scenario-based techniques [13], or rule-based methods [35]. Although advances have been made to automatically recover links, trace acquisition remains a human-intensive activity with high initial cost as also reported in case studies on industrial processes and traceability experiences [3, 18, 24, 30, 34].

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 [21]. 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 [4]. 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, needed by tool developers providing trace recovery features, and by practitioners facing the challenges of planning and managing trace recovery activities in industrial practice. Also, all current automated approach requires some human intervention. Our studies aim at a more correct assessment on the cost of this human intervention – say, if information retrieval recovers 60% of traces then the remaining 40% still need to be recovered manually and the cost of the manual recovery should not be worse than our worst case.

III. RESEARCH QUESTIONS

Our research is meant to provide worst-case data on effort and quality of trace recovery. This is ensured by using subjects not familiar with a system and providing no meaningful automation aside of basic reading technologies. We explore five research questions derived from our literature survey and our industrial experiences to better understand the human nature of trace recovery:

RQ1. 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 increases effort.* We investigate this research question in experiment 1.

RQ2. 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 investigate this research question in experiment 1.

RQ3. 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 [7]. In earlier research we explored the trade-off between trace granularity and quality [14, 17]. 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 investigate this question in experiment 2.

RQ4. 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, more intricate understanding of a class. Our expectation was that finer granularity positively impacts trace recovery quality. We investigate this research question in experiment 2.

RQ5. 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 is investigated in experiment 1.

IV. RESEARCH DESIGN

The goals of the exploratory experiments were to recover the trace links of two systems, to validate the correctness of the trace links, and to investigate the effort needed to recover requirements-to-class and requirements-to-method traces. We explore 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 support optimization of 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 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 systems is also quite large, consisting of 34 KLOC C++ code distributed in 245 classes (files) and 3490 methods. The part selected for the study consisted of 123 classes and 544 methods.

TABLE I. KEY CHARACTERISTICS OF EXPERIMENTS AND SYSTEMS.

	GanttProject (GP)	*ReactOS* (RO)
Programming Language	Java	C++
Benchmark for Correctness	key developer	key developer
Experiment 1 *(RQ1, RQ2, RQ5)* Requirements-to-Class traces	20 subjects 17 requirements 85 classes	20 subjects 16 requirements 123 classes
Experiment 2 *(RQ3, RQ4, RQ5)* Rqts-to-Method traces	48 subjects 17 requirements 788 methods	12 subjects 16 requirements 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 never able to gain system knowledge (not even during the experiment).

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 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 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 chosen 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 of course unfamiliar with the source code or implementation details – to fit the scope of this study.

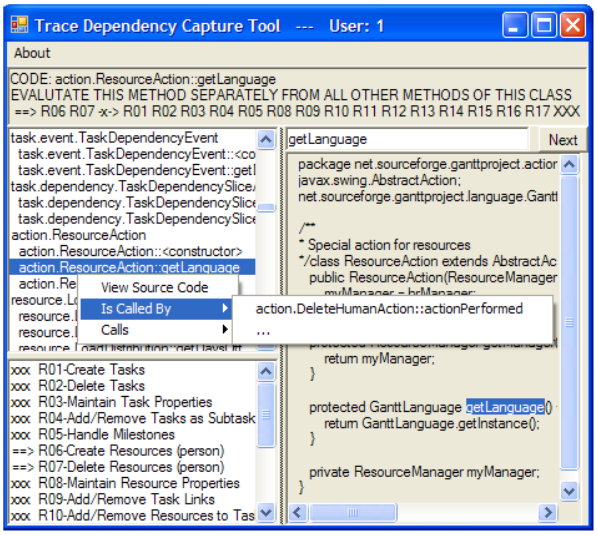

Figure 1. Trace Capture Tool.

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 a selected subset of each system and its requirements only 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 the 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 [13, 15]). While all selected classes were thus relevant to at least one requirement, trace links were still rare (only about 10% of the trace matrix had trace links). While our selecting process focused on a subset of requirements and classes, 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 to record trace links (Figure 1). The tool was purely used for data collection and did not automate any trace recovery step. 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. The tool also provided basic navigation features we considered as most relevant for trace recovery. 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 to *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 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 17x85=1445 trace/no trace votes. For the 788 GP methods in experiment 2, 17x788=13396 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 13350 votes with a redundancy of 9 votes per requirement and class. The remaining 48 subjects cast 80432 method votes with a redundancy of 6 votes per requirement and method. The low redundancy is explained by the experiment duration of 90 min per subject. Even the subset of the systems was too large for a subject to cover it entirely. A redundancy of 6 per class for 48 subjects (all subjects were given that same classes but in different order) implies that subject managed to complete 12% of the assignment in average only. This was intended and important for the experiment context because we wanted to avoid learning effects (recall that the experiment investigates the worst case which is only possible if the subjects are not familiar with the system nor gain sufficient familiarity with the system during the experiment). In order to still draw statistically significant conclusions, we thus used on many subjects.

Most votes were *no trace* votes (93%); the remaining 7% 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 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 20.073 trace/no trace votes with a redundancy of 5 votes per requirement and class. The remaining 12 subject cast 14.104 votes with a redundancy of 3 votes on a random subset per class. Even though the RO system received fewer votes in average, the large number of classes/methods covered was sufficient to draw statistically significant conclusions. Indeed, the high redundancy of GP is not necessary but since the GP experiment was conducted before the RO experiment, we lacked the benchmarks to assess the required redundancy.

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 method 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 votes were left *undefined*. Despite the lack of system knowledge, subjects 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: 80-90% 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 40-60%. It thus appears that it is often easier to rule out a trace link but that there is some gray zone where trace recovery is hard.

These results show that a very high percentage of trace links discovered by the subjects was confirmed by the benchmark data although the subjects were not familiar with the code while the developers were familiar with the software systems. Our analyses also confirmed that subjects could not have cast their votes in random and achieve such a high success rate.

V. EXPERIMENT 1: CLASS TRACES

In the first exploratory 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 13350 trace/no trace votes for the *GP* class traces and 20073 trace/no trace votes for the *RO* class traces. Since these class votes were roughly evenly distributed among all classes, we ended up with 9 votes (*GP*) and 5 votes (*RO*) per requirement and class.

A. Code complexity and effort (RQ1)

Research question 1 explores 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 believe that larger code size (LOC) and higher code complexity (McCabe's cyclomatic complexity [26]) increases trace recovery effort. We indeed found a correlation between LOC and effort (Figure 2 shows the data of GP and RO class traces). The correlation between size of classes and effort (Figure 2) is highly significant at the 1% level for both GanttProject and ReactOS with Spearman's rho being 0.408 for GP (p-value 0.00005) and .480 for RO (p-value = 0.00313).

Interestingly, GP effort per class was also significantly 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 (data omitted for brevity). The results of the RO system confirm this trend although the effort increase was not as strong. Do note, however, the logarithmic scale of the x-axis and the linear trend of the effort data. It implies that effort increases with larger classes but not linearly with class size. This indicates an economy of scale compared to class size; smaller classes are more expensive to trace than larger classes.

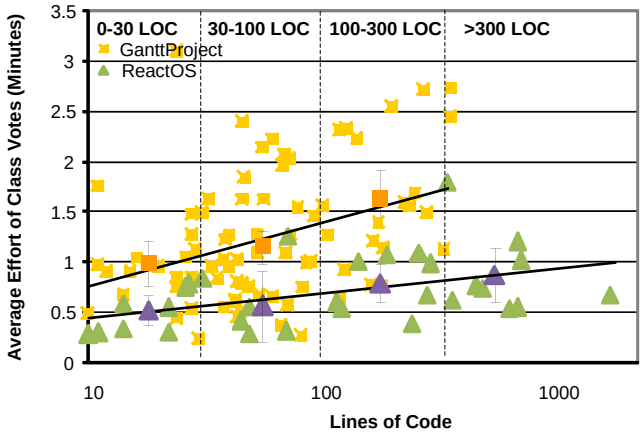

Figure 2. Code size affects class traces. Results are significant for GP and weakly correlated for RO.

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 conflicts should occur. We assessed the impact of code size and complexity (measured through McCabe's cyclomatic complexity) on trace link

quality (measured by the number of conflicts with developers). Given above observations, we would expect trace recovery to be easier for smaller classes than for larger ones. Although the average conflicts do increase with code size, the findings are clearly not as well correlated as we observed with code complexity and effort.

The correlation between size of classes and the average number of conflicts (Figure 3) is highly significant at the 1% level for GanttProject (rho 0.285, p-value = 0.0041). There also seems to be weak correlation between the size of classes and the average number of conflicts for ReactOS (rho 0.229) but we cannot claim statistical significance in this case. Again also note the logarithmic scale of the x-axis, implying that increasing class size/complexity has decreasing effect on conflicts and thus quality. This observation suggests that syntactic features of source code do not much affect trace link quality. Only the meaning and usage seems relevant.

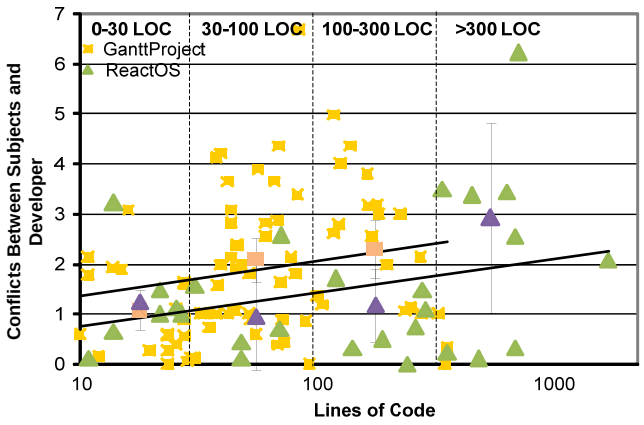

Figure 3. Code size does not affect class trace quality.

VI. EXPERIMENT 2: METHOD TRACES

Trace recovery can be done at the granularities of requirements-to-methods or requirements-to-classes. In the second exploratory experiment 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 604 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 (604 method versus 85 classes) and a fine-grained analysis of each method requires a more intricate understanding of the role of each method in the class. The 48 subjects cast 80.432 trace/no trace votes for method traces. For the 604 methods, the 48 subjects produced a coverage of roughly 6 votes per requirement and method. These votes were aggregated to make them comparable to the class traces. There were just 12 subjects available to cast the trace links between the 544 RO methods and the 16 requirements. They cast 14.104 trace/no trace votes with a coverage of roughly 2 votes per requirement and method. We could use the RO method data for considerations on single subjects (e.g. how long did a single subject need to cast a trace vote). Nevertheless, we considered a coverage of 2 votes for a requirement method pair too low to compare these results to the results of the class requirement traces with a coverage of 5 votes per requirement and class. Therefore we filtered the results and just considered requirement-method pairs with a minimum redundancy of three votes. The methods of 30 RO classes passed this threshold and were used for comparisons to experiment 1. This reduced set was still large enough to draw statistically significant conclusions.

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.

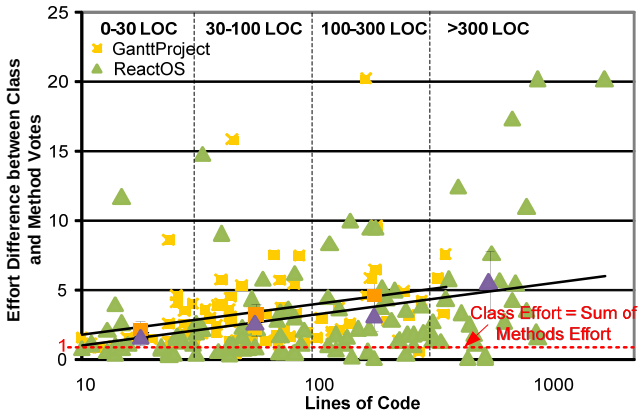

Figure 4. Comparing Method and Class Votes for GP and RO: Method votes required more effort than class votes. The factor is shown as black lines for GP (upper line) and RO (lower line). The factor increases with larger classes. The dotted line describes the situation of equal effort between class and method votes.

Figure 4 shows the effort difference between recovering method traces versus class traces. If the sum of the individual method efforts of a class exceed the class effort as a whole then it is above the nominal line (red, dashed line), otherwise it is below (equal effort yields 1). The correlation between the size and the method-to-class effort ratio (Figure 4) is highly significant at the 1% level for both GanttProject (rho .389, p-value 0,00011) and ReactOS (rho .392, p-value = 0.00005). This confirms our expectation. However, surprising is 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 produces trace links at a 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 conflicts produced). Given above observations, we already know that trace recovery was not easier for smaller, less complex trace links (Section 2) 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 87 (GP) / 45 (RO) class traces and 96 (GP) / 38 (RO) aggregated method traces, class traces and method traces agreed in 41 (GP) / 28 (RO) cases. 55 (GP) / 17 (RO) class traces were not found at the granularity of methods while 46 (GP) / 10 (RO) aggregated method traces were not found at the granularity of classes. For better understanding agreements 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. 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	1036	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

Table II summarizes our findings for both systems. We see, for example, in row 4 that of the class traces 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 methods traces not found 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 ReactOS was very similar (see right column). Surprisingly, in these cases the developers agreed slightly more with the subjects' doing. 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 significant 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 (i.e., more granularity).

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 (=effort) 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 the four buckets and the average number of conflicts of the trace votes compared to the benchmark data for each bucket. A surprising finding is that the more effort subjects spent on a given class the more likely their votes were in conflict with the benchmark. The rightmost bucket, for example, contains the conflicts of those classes where the subjects spent the least effort. This bucket with the fastest classes contains significantly fewer conflicts than the bucket with the slowest classes (95% confidence). 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 chose freely how much time to 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 yield a higher effort cost but no quality improvement either*.

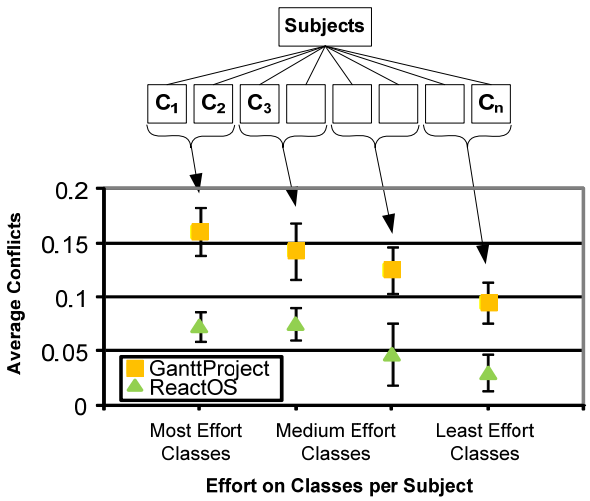

Figure 5. More effort spent on a class resulted in more conflicts. The right bucket contains conflicts of the fastest subjects for each class.

Regarding RQ5 we conclude that more effort does not yield better trace quality – neither through granularity nor through longer evaluation times. Trace recovery is fast and accurate when the assigned classes are easily 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 [37].

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 decided to use 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 (exhibiting aging effects). We thus believe that they represent typical systems found in industry that are no longer understood by its developers.

The 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 much 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. Also, we aimed at high redundancy of trace votes to further minimize this problem.

Moreover, one would expect there to be a startup phase where trace recovery is slow and a fatigue effect setting in after prolonged trace recovery. Other researchers have suggested that trace recovery should be done incrementally, in short but frequent sessions [9]. We found that subjects worked near optimial after only 20 minutes of the experiments. Our findings were not biased much by the experiment setup (data excluded for brevity).

Regarding conclusion *validity* we have computed statistical significance when analyzing the results of both experiments. While the redundancy at the level of individual classes or methods would not have been high enough for statistical significance, the high number of classes/methods investigated did make our findings statistically significant.

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 (average 90% as seen in Figure 6). 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 always, one would cast 90% correct no traces (as the students did) but 0% correct trace votes (while the students cast 50% correct trace votes). The important fact is that the students found significant numbers of correct traces and correct no traces. *The findings in this study are thus the result of ability – not luck.*

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 *GanttProject* as whole. The question is whether these classes are representative of Java classes in general? The selection of the 85 from 450 classes was 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 [22], a condition that is clearly 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 with actual developers of the system) shows that students certainly had the necessary technical skills to perform the trace recovery task.

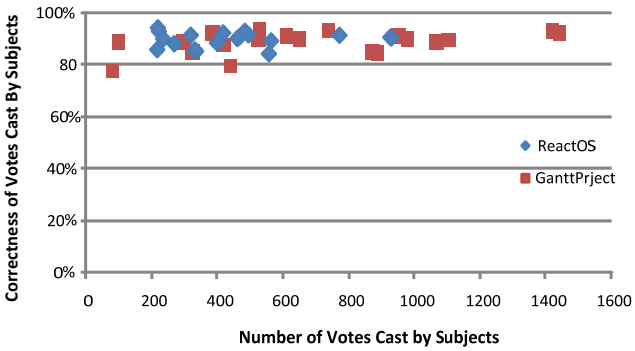

Figure 6. Performance of subjects.

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 the right figure). We can report positive correlation between the experience of subjects and the correctness of trace links was significant at the 5% level (rho .313, p-value 0.0217). We did not find a correlation between the experience of subjects and the correctness of all trace links. These findings, however, correlate weakly and are not statistically significant. Experience thus seems to matter little during trace recovery: another indication that students as subjects are well suited.

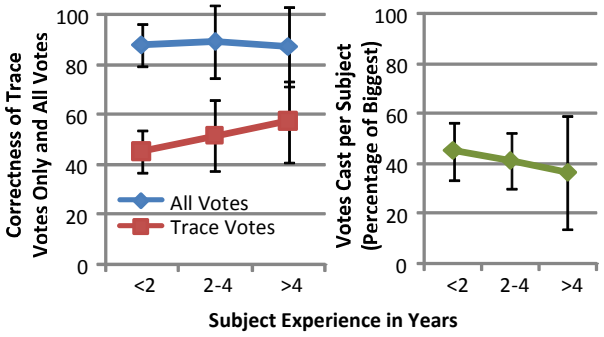

Figure 7. Experienced subjects perform better but need longer.

IX. IMPLICATIONS FOR PRACTIONERS

For practitioners, our study reveals interesting facts about the cost and quality of trace capture (worst case):

- Trace recovery costs in average 0.5-2 minutes depending on code size with the cost increasing non-linearly (twice the code size is not twice the cost)
- 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 (50% correct trace votes and 95% correct no trace votes). Yet, the quality appears to be weakly correlated to the code size and subject experience.
- Trace recovery of method traces costs between 2-6 times as much as class traces – again with the cost increasing non-linearly. Most interesting, however, the quality of method traces is not better than that of class traces – even though they require more effort. We generally found that trace effort does not correlate with trace quality which is a surprising observation.

X. CONCLUSIONS

We presented the results of two exploratory experiments on recovering trace links between requirements and classes (experiment 1) as well as requirements and methods (experiment 2). 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 trace capture 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 in average a random set of 24 classes only (15% of the 85 classes covering the core functionality which is less than 3% of the total 516 classes). This clearly did not allow a reasonably complete 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 [10, 31]. Despite of these constraints, subjects were able to identify meaningful trace links. Trace quality was surprisingly high (with *no trace* links easier to determine correctly compared to *trace* links).

In context of our research questions, the findings are:

Regarding research question RQ1, the data indicates that code complexity increases 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, within any given class, more effort does not mean better quality.*

Regarding research question RQ2 our analyses reveal that there is only a weak 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 fragments. 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. This also suggests that trace recovery does not suffer greatly from scalability problems were larger, more complex classes would become less recoverable.

Regarding research question RQ3 our experiment showed 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 also explored trace effort and quality differences in various phases of working sessions as only little is known about the time it takes to get up to speed and about the optimal duration of trace sessions. Data suggests that trace recovery should be done incrementally, in short but frequent sessions. We found that trace recovery has a short learning phase (<20min), reaches optimum quickly (<60min) but suf-

fers from a fatigue effort (>90min). The data also suggests that trace recovery could be fairly easily split into multiple sessions, done by different people, incrementally. This is consistent with recent research results that suggest incremental over one-shot trace recovery [9].

We got surprising results regarding RQ5. A higher tracing effort does not imply better quality. Data indicates that trace link recovery falls into two categories: fast and accurate or slow and inaccurate. At this point, the only remedy against bad quality seems to be redundancy by assigning multiple subjects to any given class. If affordable, it improves quality because we observed that subjects did not uniformly perceive the same classes as difficult. It seems that a long classification time is indicative of uncertainty, leading to decreased precision.

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 optimize features for trace recovery automation. 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 the first, 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. *Our benchmark is intentionally based on a worst case scenario – in part because of simplicity due to the large number of factors involved but also because if it can be shown that an application of a trace is cost-effective even in a worst-case situation then the application is truly cost effective in general.*

REFERENCES

[1] Aizenbud-Reshef, N., Nolan, B.T., Rubin, J. and Shaham-Gafni, Y. Model Traceability. IBM Systems Journal, 45 (3). 515-526.

[2] Antoniol, G., Caprile, B., Potrich, A. and Tonella, P. Design-Code Traceability Recovery: Selecting the Basic Linkage Properties. Science of Computer Programming, 40 (2-3). 213-234.

[3] Asuncion, H.U., Francois, F. and Taylor, R.N. An end-to-end industrial software traceability tool 6th Int'l ESEC/FSE conference, ACM, Dubrovnik, Croatia, 2007.

[4] Bianchi, A., Visaggio, G. and Fasolino, A.R. An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models 8th Int'l Workshop on Program Comprehension, IEEE CS, 2000.

[5] Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R. and Romanova, E. Best Practices for Automated Traceability. Computer, 40 (6). 27-35.

[6] Cleland-Huang, J., Chang, C.K. and Christensen, M. Event-Based Traceability for Managing Evolutionary Change. IEEE TSE, 29 (9). 796-810.

[7] Cleland-Huang, J., Zemont, G. and Lukasik, W. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability 12th IEEE Int'l RE Conference, IEEE CS, 2004.

[8] Dagenais, B., Breu, S., Frederic Weigand, W. and Robillard, M.P. Inferring structural patterns for concern traceability in evolving software 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering, ACM, Atlanta, GE, 2007.

[9] De Lucia, A., Oliveto, R. and Tortora, G., IR-Based Traceability Recovery Processes: An Empirical Comparison of "One-Shot" and Incremental Processes. 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering, (2008), L'Aquila, Italy, 39-48.

[10] de Lucia, A., Oliveto, R., Zurolo, F. and Penta, M.d. Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment 14th IEEE Int'l Conf. on Program Comprehension, IEEE CS, 2006.

[11] Deng, M., Stirewalt, R.E.K. and Cheng, B.H.C. Retrieval by Construction: a Traceability Technique to Support Verification and Validation of UML Formalizations. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), 15 (5). 837-872.

[12] Duan, C. and Cleland-Huang, J. Clustering support for automated tracing 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering, ACM, Atlanta, GE, 2007.

[13] Egyed, A., A Scenario-Driven Approach to Traceability. 23rd Int'l Conf. on Softw. Eng., (Toronto, 2001), 123-132.

[14] Egyed, A., Biffl, S., Heindl, M. and Grünbacher, P., Determining the Cost-Quality Trade-Off for Automated Software Traceability. 20th ACM/IEEE Int'l Conf. on Automated Software Eng., 2005.

[15] Egyed, A., Binder, G. and Grünbacher, P., STRADA: A Tool for Scenario-based Feature-to-Code Trace Detection and Analysis. 29th Int'l Conf. on Software Engineering, (St. Louis, Missouri, 2007), IEEE CS, 41-42.

[16] Egyed, A. and Grünbacher, P., Automating Requirements Traceability: Beyond the Record & Replay Paradigm. 17th IEEE Int'l Conf. on Automated Software Engineering, (Edinburgh, 2002), IEEE CS, 163-171.

[17] Egyed, A., Grünbacher, P., Heindl, M. and Biffl, S., Value-Based Requirements Traceability: Lessons Learned 15th IEEE Int'l Requirements Engineering Conference, (New Delhi, India, 2007), IEEE CS, 115-118.

[18] Gotel, O. and Finkelstein, A., Extended Requirements Traceability: Results of an industrial case study. 3rd Int'l Symposium on Requirements Engineering, (1997), 169-178.

[19] Gotel, O.C.Z. and Finkelstein, A.C.W., An Analysis of the Requirements Traceability Problem. 1st Int'l Conf. on Requirements Eng., (1994), 94-101.

[20] Hayes, J.H. and Dekhtyar, A. Humans in the traceability loop: can't live with 'em, can't live without 'em 3rd Int'l Workshop on Traceability in Emerging Forms of Software Engineering, ACM, Long Beach, CA, 2005.

[21] Hayes, J.H., Dekhtyar, A., Sundaram, S.K. and Howard, S. Helping Analysts Trace Requirements: An Objective Look 12th IEEE Int'l Requirements Engineering Conf., IEEE CS, 2004.

[22] Höst, M., Regnell, B. and Wohlin, C. Using Students as Subjects: A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. Empirical Software Engineering, 5. 201-214.

[23] Koschke, R. and Quante, J. On dynamic feature location Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM, Long Beach, 2005.

[24] Lindvall, M. and Sandahl, K. Practical implications of traceability. Softw. Pract. Exper., 26 (10). 1161-1180.

[25] Marcus, A. and Maletic, J.I. Recovering documentation-to-source-code traceability links using latent semantic indexing 25th Int'l Conf. on Software Engineering, Portland, 2003.

[26] McCabe, T.J. A Complexity Measure. IEEE TSE, 2 (4). 308-320.

[27] Muccini, H., Bertolino, A. and Inverardi, P. Using Software Architecture for Code Testing. IEEE TSE, 30 (3). 160-171.

[28] Murphy, G.C., Notkin, D. and Sullivan, K., Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. 3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering, (1995), New York, 18-28.

[29] Murta, L.G.P., van der Hoek, A. and Werner:, C.M.L. Continuous and automated evolution of architecture-to-implementation traceability links. Autom. Softw. Eng., 15 (1). 75-107

[30] Neumüller, C. and Grünbacher, P., Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. 21st IEEE Int'l Conf. on Automated Software Eng., (2006), 145-156.

[31] Parnas, D.L., Software Aging. Int'l Conf. on Software Engineering, (1994), 279-287.

[32] Pohl, K. PRO-ART: Enabling Requirements Pre-Traceability 2nd Int'l Conference on Requirements Engineering, 1996.

[33] Ramesh, B. and Jarke, M. Toward Reference Models for Requirements Traceability. IEEE TSE, 27 (4). 58-93.

[34] Ramesh, B., Stubbs, L.C. and Edwards, M. Lessons Learned from Implementing Requirements Traceability. Crosstalk: Journal of Defense Software Engineering, 8 (4). 11-15.

[35] Spanoudakis, G., Zisman, A., Perez-Minana, E. and Krause, P. Rule-based generation of requirements traceability relations J. Systems and Software, 72 (2). 105-127.

[36] Walker, R.J., Baniassad, E.L. and Murphy, G.C., An initial assessment of aspect-oriented programming Proceedings of the 21st international Conference on Software Engineering (Los Angeles, 1999), ACM, 120-130.

[37] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B. and Wesslén, A. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, 2000.

[38] Yue, T., Briand, L.C. and Labiche, Y. Automated Traceability Analysis for UML Model Refinements. Journal of Information and Software Technology, 51 512-527.