Value-Based Requirements Traceability: Lessons Learned

Alexander Egyed
Teknowledge Corp.
Marina Del Rey, USA
aegyed
@teknowledge.com

Paul Grünbacher
Johannes Kepler Univ.
A-4040 Linz, Austria
Paul.Gruenbacher
@jku.at

Matthias Heindl
PSE Siemens Austria
A-1100 Vienna, Austria
matthias.a.heindl
@siemens.com

Stefan Biffl
Vienna Univ. of Techn.
A-1040 Vienna,Austria
Stefan.Biffl
@tuwien.ac.at

Abstract

Software development standards demand requirements traceability without being explicit about the appropriate level of quality of trace links. Unfortunately, long-term trace utilizations are typically unknown at the time of trace acquisition which represents a dilemma for many companies. This paper suggests ways to balance the cost and benefits of requirements traceability. We present data from 3 case studies. Lessons learned suggest a traceability strategy that (1) provides trace links more quickly, (2) refines trace links according to user-definable value considerations, and (3) supports the later refinement of trace links in case the initial value considerations change.

1. Introduction

Trace links define dependencies among key software artifacts such as requirements, design elements, and source code. They support engineers in understanding complex software systems by analyzing properties such as completeness, conflicts, or coverage [3][4]. Traceability is nowadays demanded by numerous standards, such as ISO 15504 or the CMMI.

Capturing trace links requires a significant effort even for moderately complex systems [4]. While some automation exists, capturing traces remains a largely manual process of quadratic complexity. Even worse trace links degrade over time and have to be maintained continuously to remain useful over time.

It would be uneconomical, however, to capture trace links completely and correctly as even less-than-perfect trace links yield benefits. Engineers need to consider both the near-term and long-term utilization needs of trace links. This paper presents lessons learned from three case studies. The lessons suggest that one should first *identify trace links quickly and completely on a coarser level of granularity* and then refine them according to some user-definable value consideration (i.e., predicted utilization needs).

The traceability life cycle includes four tasks:

Acquisition. Software engineers create trace links between requirements and other artifacts such as design elements, or source code either manually or with the help of tools.

Utilization. Software engineers consume trace links to support change impact analysis, requirements dependency analysis, etc. One needs to distinguish short-term utilization (e.g., determining test coverage in later project stages) and long-term utilization (e.g., a particular change request years later).

Maintenance. Software engineers continuously revisit and update trace links as the system and its artifacts evolve. Trace maintenance ensures that the quality of trace links does not degrade.

Enhancement. Software engineers improve the quality of trace links (e.g., their completeness or correctness) if the quality is insufficient for the intended utilization.

Better tools, more capable engineers, more calendar time, or better documentation are certainly helpful to improve the quality and to reduce the cost of traceability. But two fundamental problems remain:

Finding the right level of trace quality with finite budget. Even if developers have a quality threshold in mind, it is not obvious whether the allocated budget is sufficient for the planned traceability task. For example, it is not obvious that improving trace links is cost-efficient, i.e., the benefits gained through trace utilization are offset by the added cost of producing better trace links.

Increasing the quality of trace links comes at an increasingly steep price. Trace acquisition suffers from a diseconomy of scale where low-quality trace links can be produced fairly quickly and economically while perfection is expensive and hard to determine.

Engineers performing traceability tasks have insufficient time to complete the task in a complete, correct, and consistent manner. They can use two fundamental strategies to deal with the problem: (1) “Brute force”, i.e., trying to generate the trace links for the complete system in the limited time available. Obviously the insufficient time will have a negative impact on the correctness of trace links and their later utilization.

(2) “Selective”, i.e., trying to find the most valuable traces driven by an explicit or implicit *value-based strategy* such as easy-things-first, gut feeling, business importance, or predicted future utilization until running out of resources. As a result some parts of the system will have trace links of reasonable quality while other trace links will be missing or incorrect. This can limit or even preclude future utilization.

2. Lessons Learned in Three Case Studies

We derive the lessons for value-based traceability from three case studies: The open-source ArgoUML[1] tool, an industrial route-planning application from Siemens Corporation, and an on-demand movie player. ArgoUML is an open-source software design tool supporting the Unified Modeling Language (UML). The Siemens route-planning system supports efficient public transportation in rural areas with modern information technologies. The Video-On-Demand system[2] is a movie player allowing users to search for movies and playing them.

2.1. Adjusting Granularity

Granularity is the level of precision of trace links (e.g., requirements to packages vs. requirements to classes). The needs of the techniques that utilize traces links normally drive this decision. The benefit of adopting coarse-grained trace links is better coverage and higher quality of trace links at lower costs. However, there is a sacrifice: Low granularity trace links are not as precise and useful during trace utilization. Adjustment of granularity provides a cheap way for experimenting with the correctness and completeness of traces.

The trace links in the three case studies were between requirements and source code. We analyzed the granularity trade-off for the three systems. Cost was measured in terms of the effort required and the input quantity generated. We considered the following three levels of granularity: requirements-to-methods, requirements-to-classes, and requirements-to-packages. Quality was measured in terms of the number and percentage of false positives. In particular, for each case study system we analyzed the impact of trace acquisition on the quality of the generated trace links. As a baseline, we took the level of false positives produced on the most detailed level of granularity (i.e., requirements-to-methods). The analysis compared how a reduction of granularity resulted in a higher number of false positives (note that a reduction in granularity does not cause false negatives – missing trace links).

[1] http://argouml.tigris.org/
[2] http://peace.snu.ac.kr/dhkim/java/MPEG

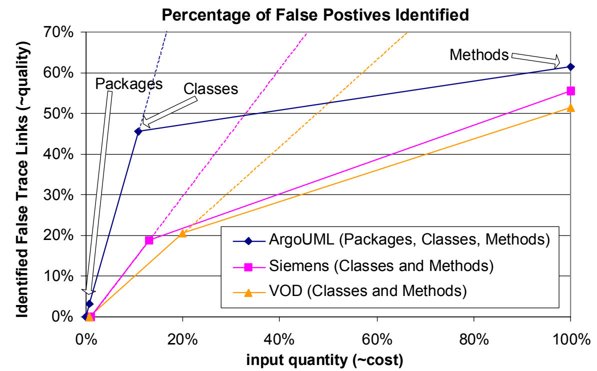


Figure 1. Decreasing number of false positives with increasing level of detail.

Figure 1 presents our findings for the three levels of granularity and the three case study systems. For example, the ArgoUML system consisted of 49 packages, 645 classes, and almost 6,000 methods. The number of trace links captured at the granularity of Java classes was thus only one-tenth the order of magnitude compared to the quantity at the granularity of methods. This reduction in input quantity also led to a three-fold reduction compared to the effort needed to generate the coarser-grained trace links. However, this saving came at the expense of trace quality. Figure 1 also shows the quality drop relative to the total number of traces. We found that trace links at the granularity of classes had 16% more false positives compared to the trace links at the granularity of methods. This effect was much stronger on the granularity of packages which had over 40% more false positives with another ten-fold reduction in input quantity (20/30% more false positives on the level of classes and 55% more false positives on the level of packages – no packages were defined for the movie player). *Our data strongly indicates that there is a decreasing marginal return on investment (ROI) with finer-grained input. Indeed, the data strongly suggest that the granularity of classes provides the best cost/quality trade-off.* Adjusting the level of granularity can be used as a cost saving measure and some techniques that utilize trace links would still produce reasonable results.

2.2. Value-based Enhancements

Granularity-based trace links can save substantial cost during trace maintenance and enhancement. However, the resulting across-the-board quality reduction may not be acceptable. While engineers may be willing to sacrifice some benefits to save cost, we believe that such a process must be guidable. In the following, we discuss a value-based extension to granularity-based trace acquisition, maintenance, and enhancement: Value-based software engineering [1] invests effort in areas that matter most. A value-based approach relies on considering stakeholder value-propositions to right-

size the application of a software engineering technique. The definition of value depends largely on the domain, business context and company specifics. Our approach does not prescribe a particular value function. In essence, engineers can place value directly on trace links (i.e., this link is important) or indirectly on the artifacts they bridge (i.e., this requirement is important and consequently also its trace link to code).

2.2.1 Understanding the Diseconomy of Scale

Intuition says that if only half the trace links are important then only half of them need to be refined to a finer level of granularity – thus saving 50% of the cost. This intuition is misleading as requirements map to large portions of source code and each piece of code may be related to multiple requirements. Above we presented details on the 38 requirements-to-code traces for ArgoUML. These 38 requirements covered 4,752 methods (roughly 80% of the ArgoUML source code) with the average trace covering 248 methods.

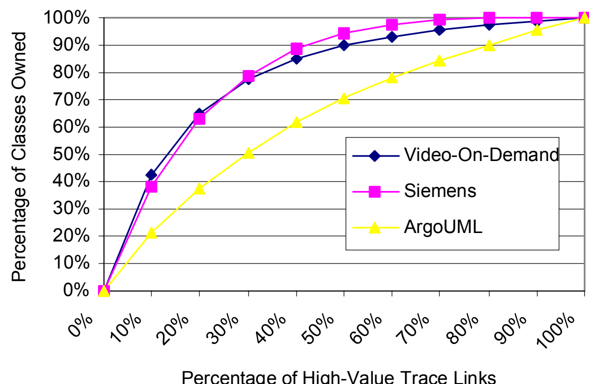


Figure 2. Diseconomy by Enhancing Classes belonging to a High-Value Trace Link.

Trace acquisition is usually not done by taking a requirement and guessing where it might be implemented. Typically, trace acquisition iterates over the source code, one class/method at a time, and reasons to which requirement(s) it belongs. For the ArgoUML system, we found that a class was related to 3.2 requirements in average. If only one of these three requirements was important then the class would need to be refined. The likelihood for this increased non-linearly. Figure 2 depicts the percentage of classes that were traced to by at least one high-value trace link in relationship to the percentage of high-value trace links. The cost is normalized across all three case studies. It can be seen that the cost varies somewhat although it is similarly shaped. The *x*-axis depicts the percentage of high-value trace links. A high number of high-value trace links increases the number of classes they own collectively. However, we also observe a diseconomy of scale. For example, if 40% of the ArgoUML trace links are of high value then half of its classes (i.e., more than 40%) are owned by them. Consequently, 50% of the classes need to be refined. The other two case studies behaved much worse. In both cases, 40% of the high-value trace links owned almost 80% of the classes. This diseconomy of scale seems to invalidate the benefits of value-based trace acquisition and shows why the simple selective strategy is not desirable. In the case of the Siemens and VOD systems, the cost for trace enhancement for 40% high-value requirements is almost as high as doing the enhancement for *all* requirements.

2.2.2 Enhancing Common Classes

Fortunately, there is also a positive effect that counters this diseconomy of scale. We made the trivial assumption that every class owned by a high-value trace link must be refined to the granularity of methods. This is however not necessary. Figure 3 depicts four trace links: two high-value trace links covering requirements 1 and 2; and two low-value trace links covering requirements 3 and 4. Each circle represents the set of classes traced to by each requirement. These requirements "share" some classes, i.e., their traces overlap in their common use of classes as indicated by the intersecting circles but also own classes they do share with other requirements [2].

Which of the classes in the various areas (overlapping or not) in Figure 3 must be refined to a finer level of granularity? We distinguish five areas: (1) classes owned by a single high-value requirement; (2) classes owned by a single low-value requirement; (3) classes shared among high-value requirements; (4) classes shared among low-value requirements; and (5) classes shared among multiple requirements including one high-value requirement (if there are multiple high-value requirements than area 3 applies).

Obviously, classes owned by low-value requirements (area 2) or shared among low value requirements (area 4) should not be enhanced. However, even classes owned by single high-level requirements (area 1) do not need to be enhanced. We discussed previously that coarse granularity is correct and complete. Thus if a class is owned by a single requirement then all its methods must be owned by this artifact (i.e., if the class is not shared then its methods cannot be shared either). However, classes owned by multiple high-level requirements (area 3) must be enhanced because we cannot decide what methods are owned by the one requirement versus the other.

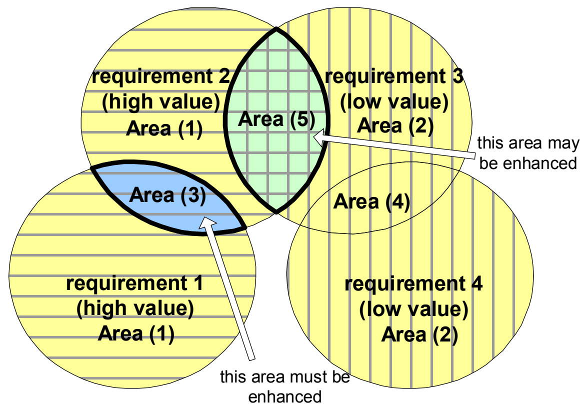


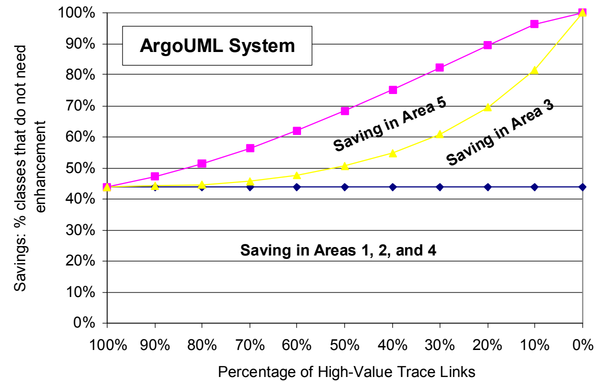
Figure 3. Detailed Granularity Only Necessary for Overlaps Involving Higher-Value Trace Links.

Only overlaps between a single high-level class and one or more low-level classes (area 5) represent a gray zone. Most techniques do not benefit from the enhancement of area 5. In those cases, defining area 5 for one requirement but no other is a waste also.

These observations lead to substantial savings with no loss in quality. Figure 4 depicts the results for the ArgoUML case study (top) and the VOD case study (bottom); the Siemens results are similar. Over 45% of the 645 classes of the ArgoUML were owned by single requirements (areas 1, 2, and 4). These classes needed not be refined to a finer level of granularity. This resulted in an instant saving of 12-45% effort depending on the case study. This saving was independent of the percentage of high-value trace links.

In addition, depending on the percentage of high-value trace links, we saved on the overlapping areas 3 and 5 (itemized separately). For example, if 40% of the trace links were of high value then an additional 39% of input effort was saved (in both case studies) because many of the overlapping areas did not require fine-grained input. *In our experience between 15-50% of trace links are typically of high value. Based on our case studies, this translates to 30-70% savings during enhancement compared to a value-neutral approach*. This extra saving does not reduce trace link quality.

So not only trace maintenance benefits from complete, coarse-grained trace acquisition on the granularity of classes. Even the enhancement of trace links during trace acquisition benefits from it because it piggybacks from the results obtained on the coarser granularity to decide where to refine. From ArgoUML's 645 classes, only 134 needed to be refined to the granularity of methods. Given that there were in average 9.2 methods per class in the ArgoUML system 1,232 methods needed to be looked at in more detail compared to 6,000 methods in case of a value-neutral approach, a reduction of 80%.



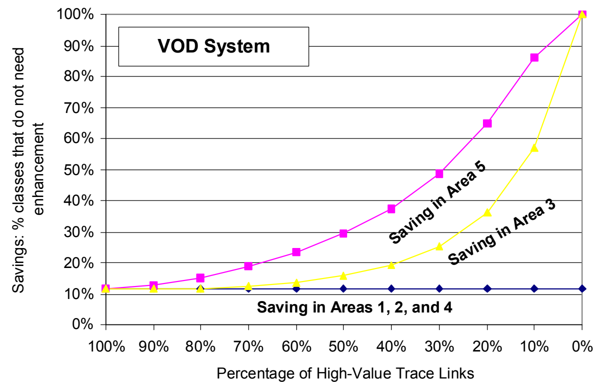


Figure 4. Effort Saved due to Value Considerations in the Source Code (top: ArgoUML, bottom: VOD)

3. Conclusions

In this paper we presented three case studies to support a value-based approach to software traceability, i.e., spending the money where it matters most (value); exploring trace links incrementally based on an initial, complete base of trace links; and considering trace utilization, maintenance, and enhancement. Neglecting these lessons will lead to higher cost and inappropriate trace links. Ad-hoc trace generation may have some immediate benefits but is bound to result in more disadvantages over the course of the software development life cycle and its maintenance.

References

[1] Biffl, S., Aurum, A., Boehm, B. W., Erdogmus, H., Grünbacher, P.: „*Value-based Software Engineering*." Springer Verlag, 2005.

[2] Egyed A.: A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE TSE* 29(2), 2003, 116-132.

[3] Gotel, O. C. Z. and Finkelstein, A. C. W.: "An Analysis of the Requirements Traceability Problem," *Proc. ICRE* 1994, pp.94-101.

[4] Ramesh, B., Stubbs, L. C. and Edwards, M. Lessons learned from implementing requirements traceability. Crosstalk 8(4):11–15, 1995.