Value-Based Requirements Traceability: Lessons Learned

Alexander Egyed[1], Paul Grünbacher[1], Matthias Heindl[2], and Stefan Biffl[3]

[1] Institute for Systems Engineering and Automation, Johannes Kepler University, A-4040 Linz, Austria
{ae,pg}@sea.uni-linz.ac.at
[2] Siemens IT Solutions and Services, Siemens Austria, Gudrunstraße 11, A-1100 Vienna, Austria
matthias.a.heindl@siemens.com
[3] Institute for Software Technology and Interactive Systems, Vienna Univ. of Techn., A-1040 Vienna, Austria
Stefan.Biffl@tuwien.ac.at

Abstract. Traceability from requirements to code is mandated by numerous software development standards. These standards, however, are not explicit about the appropriate level of quality of trace links. From a technical perspective, trace quality should meet the needs of the intended trace utilizations. Unfortunately, long-term trace utilizations are typically unknown at the time of trace acquisition which represents a dilemma for many companies. This chapter suggests ways to balance the cost and benefits of requirements traceability. We present data from three case studies demonstrating that trace acquisition requires broad coverage but can tolerate imprecision. With this trade-off our lessons learned suggest a traceability strategy that (1) provides trace links more quickly, (2) refines trace links according to user-defined value considerations, and (3) supports the later refinement of trace links in case the initial value consideration has changed over time. The scope of our work considers the entire life cycle of traceability instead of just the creation of trace links.

Keywords: Requirements engineering, software traceability, value-based software engineering.

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 identifying where artifacts are implemented [12]. A significant body of work has been published on software traceability and its usefulness, particularly on requirements traceability [12,19]. Traceability has also made its way into a number of software engineering standards and initiatives, such as ISO 15504 and the CMMI that mandate or recommend traceability as 'best practice'.

K. Lyytinen et al. (Eds.): Design Requirements Workshop, LNBIP 14, pp. 240–257, 2009.

Over the last couple of years we have been collaborating in the area of requirements traceability with several industry partners in the US and Europe. These included large organizations such as Siemens Austria [13], Boeing Company, and NASA but also very small companies such as geDV [17]. We have seen that companies introducing traceability techniques face significant challenges. Capturing trace links requires a significant effort even for moderately complex systems [19]. While some automation exists for trace capture it still remains a mostly manual process of non-linear complexity (i.e., if m requirements trace to n pieces of source code then there are m times n potential trace links). Even worse, once generated, trace links degrade over time as the software system evolves. While trace links are often utilized immediately, they still have to be maintained continuously thereafter to remain useful over time. Maintaining trace links after changes to the system is a continuous, manual task, also of quadratic complexity.

In practice, engineers rarely capture trace links completely because of the uncertainty of their later usage. Engineers thus attempt to predict what trace links will likely be needed in the future and concentrate on these. If the trace utilization is planned for the near term their predictions tend to be good. There is, however, a significant uncertainty concerning long-term utilization. Engineers thus often have to err on the side of perfection (i.e., producing links that are not needed later) or on the side of incompleteness (i.e., omitting links that are needed later). Both errors can be costly. The practical alternative, i.e., to delay trace acquisition and to create trace links upon request, is typically infeasible because the manual generation and maintenance of trace links hinges on the engineers' ability to recollect necessary facts. Asking for traceability information long after the fact is like asking for documentation as soon as somebody decides to read it.

The dilemma of requirements traceability is thus that engineers must consider both the near-term and long-term utilization needs of trace links.

This chapter presents lessons learned from three case studies. The lessons suggest that a traceability strategy should first identify trace links quickly and completely on a coarser level of granularity and then refine them according to some user-defined value consideration (i.e., based on predicted utilization needs). Engineers may still err during the refinement – if value considerations change or turn out to be incorrect – however, our lessons learned indicate that engineers are able to recover from such errors more quickly if they can rely on a complete set of coarser-grained trace links. Software maintenance costs represent a significant share of the total software cost so our analysis tackles an important industry problem [5]. This chapter builds on an initial workshop paper [4] describing the proposal for this work and a short conference paper discussing some of the basic points [11]. The novel contributions in this chapter are the detailed data of the three case studies together with lessons learned that can be used directly by practitioners in their projects.

The chapter is structured as follows: Section 2 discusses related work. Section 3 explores and illustrates the problem in greater detail. Section 4 presents lessons learned in the three case studies and detailed analyses of data. Section 5 discusses results. We round out the chapter with conclusions and an outlook on further work.

2 Related Work

Numerous commercial and research approaches are available that support requirements traceability. Most approaches focus on the acquisition and management of trace links: Tool support exists in two forms: (1) Tools for capturing and managing trace links provide a structured way of defining trace links and keeping track of missing or potentially outdated links. Many commercial tools primarily focus on these aspects of recording and replaying trace links. The tools have very limited capabilities to automate the definition of trace links, which remains a manual and error-prone activity. (2) Tools for automating the acquisition of trace links are typically based on providing some basic facts initially followed by some deductive reasoning to fill-in missing pieces. Examples of approaches in this category are based on scenarios [9] or information retrieval techniques [14]. Such tools are sometimes based on heuristics and neither complete nor correct results can be guaranteed.

Despite existing tool support traceability remains costly and complicated. There are typically no guarantees of completeness or correctness (errors and omissions). Developers can neither fully trust automatically generated trace links nor manually defined ones.

Traceability approaches typically do not provide explicit support for trace utilizations such as impact or coverage analysis. They rather provide general-purpose features to create reports or query traceability information. Researchers have been proposing techniques to improve support for important tasks such as analyzing change impacts [1] or understanding the conflict and cooperation among requirements [10].

There is very little literature on the quality implications of trace links. This is partly because there are many applications that benefit from trace links with different cost/quality trade-offs. An exception is [2]: Bianchi *et al.* show that the effectiveness of maintenance can be improved by varying the degree of granularity of the traceability model. A reason for the lack of research results lies in the fact that it is unknown in advance which trace links will be used.

Murphy *et al.* have explored the idea of "good enough" techniques in the context of closing the gap between design and implementation. Their reflexion model technique helps engineer evolving a structural mental model of a system. As soon as this model is "good enough" it can be used for reasoning about tasks such as design conformance, change assessment, and experimental reengineering.

Recently, several publications appeared in the area of value-based software engineering [3,6]. This thread of research provides a new perspective on balancing cost and benefits of software engineering techniques. An initial cost-benefit analysis for traceability is reported in [13]. Cleland-Huang *et al.* have also started exploring the economic aspects of traceability [7]. In [15], Lindvall and Sandahl show in a case study that tailoring of traceability models is essential in practice.

The literature review reveals that many traceability approaches emphasize isolated aspects (e.g., trace generation) but do not consider the full life cycle "end-to-end" traceability. Furthermore, the impact of different quality levels of trace links is not yet well understood. There is still no systematic way for

understanding the value contribution of trace links and for dealing with cost-quality trade-offs.

3 Problem Illustration

To illustrate the problems in more detail we take a look at the life cycle of software traceability. This life cycle includes in essence four tasks:

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

Utilization. Software engineers consume trace links in tasks such as change impact analysis, requirements dependency analysis, etc. It is useful to distinguish between 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 various artifacts (requirements, design elements, code, etc.) are being changed. Trace maintenance ensures that the quality of trace links does not degrade.

Enhancement. Software engineers improve the quality of trace links (i.e., increasing completeness or correctness) in case their quality is insufficient for the intended utilization.

Better tools, more capable engineers, more calendar time, or better documentation are certainly helpful in improving the quality and reducing the cost of traceability in any or all tasks. But, in essence these measures do not mitigate the following fundamental problems:

Finding the right level of trace quality with finite budgets. Even if developers have some 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 as 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 very hard to achieve and determine.

Traceability planning under uncertainty about future utilization needs. We cannot know which trace links at which level of quality will be needed in the future as detailed knowledge about applications utilizing them is not available at the time of their creation.

An engineer performing a traceability task typically faces a situation where the time available to complete the task is much shorter than the time required performing the task in a complete, correct, and consistent manner. Basically, the engineer has two fundamental strategies to deal with the problem:

"Brute force", i.e., trying to generate the trace links for the complete system in the limited time available. Obviously this will have some negative impact on

Table 1. Impact of "Brute force" vs. "elective" strategies on traceability tasks

Task	*Brute force strategy*	*Selective strategy*
Acquisition	Full coverage but many incorrect links.	Correct links but small coverage (area and/or depth).
Utilization	Utilization hampered by erroneous feedback caused by incorrect links. Similar performance for short-term and long-term utilization.	Good utilization limited to available links. For other utilizations problematic or infeasible. Likely emphasizing on short term utilization.
Maintenance	Hampered by incorrect links (lack of trust in trace links).	Limited to smaller set of available trace links.
Enhancement	Hampered by incorrect links (lack of trust in trace links).	Very hard due to later unfamiliarity.

the correctness of trace links and their later utilization if the allocated time is insufficient as is often the case.

"Selective", i.e., trying to achieve the most valuable traces until running out of resources driven by an explicit or implicit value prioritization strategy such as easy-things-first, gut feeling, or predicted future utilization. As a result of applying the strategy 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 in some cases even preclude future utilization.

Table 1 summarizes the impact of the two strategies on the traceability tasks discussed earlier.

The brute force strategy negatively affects trace utilization because of the higher degree of incorrectness. While there are typically some errors in trace links due to the complexity involved in generating them, the brute force strategy worsens this situation. Trace maintenance also suffers from higher incorrectness because it builds on the presumed correct trace links. Trace enhancement is, however, unnecessary. The selective strategy is better but failure in predicting trace utilization needs will make enhancement difficult. This is less of a problem for near-term utilization because the engineers are still available and knowledgeable to recover missing trace links. However, the selective strategy suffers immensely during long-term utilization because recovering trace links later is very difficult (i.e., if engineers moved on or forgot necessary details). The most serious drawback of these two approaches is the inability to recover from missing or incorrect traces.

4 Lessons Learned in Three Case Studies

We describe lessons learned based on data from three case studies. The purpose of the lessons is to guide practitioners to define a traceability strategy and to understand the expected benefits. Our intention is not to propose a concrete approach or improve upon a specific tool or traceability technique. Rather, we demonstrate that value-based software engineering techniques can have an

Table 2. Case studies and their context

System	*Development context*
ArgoUML	UML modeling tool; distributed developers; open-source development
Siemens Route Planning	Route-planning application for public transportation; industrial developer team
Video on demand	Movie player; single developer

impact in very practical terms. We will discuss what trace links to generate at which time and at what level of detail. We will not discuss how to generate them. Any existing guidable process or tool should be able to adopt our lessons learned.

We derive the lessons from three case studies: The open-source ArgoUML tool [18], 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 is a movie player allowing users to search for movies and playing them [8]. We chose these systems because they cover a range of different development contexts (open source vs. industrial), application characteristics (large vs. small), and domains (see Table 2).

4.1 Reducing Granularity

Lessons learned:

- *Save traceability effort by reducing granularity.*
- *Requirements-to-class-level granularity provides better value for money compared to requirements-to-package-level and requirements-to-method-level granularity.*
- *Focus on completeness and correctness first – these are essential for follow up tasks such as maintenance and enhancement.*
- *Reducing granularity reduces the benefits of trace utilization.*

We learned in the case studies that combining value-based and granularity-based trace acquisition is a good strategy. Granularity is the level of precision of a trace link (e.g., requirements to packages vs. requirements to classes). Adjustment of granularity can provide a cheap and quick way of exploring correct and complete requirements-to-code traces. We will see that granularity-based trace analysis may be imprecise but it can be computed much more efficiently without sacrificing correctness and completeness.

The trace links considered in the three case studies were between requirements and source code. In some development contexts, engineers might define trace links to the granularity of methods (e.g., Java methods in case of the ArgoUML system). Sometimes, engineers might even decide to define trace links to

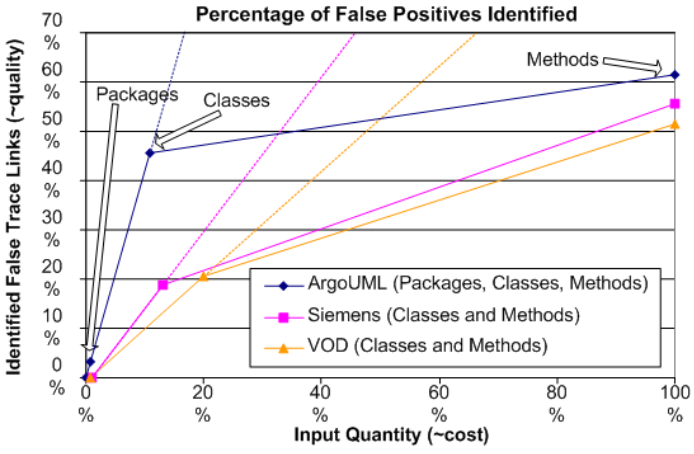

Fig. 1. Decreasing number of false positives with increasing level of detail

individual lines of code (e.g., when exploring crosscutting concerns in safety critical systems). 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. We have found, however, a remedy to this issue which will be discussed later.

We analyzed the granularity trade-off for the three case study 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).

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 quantity 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 obviously also reduced the effort spent: we observed a three-fold reduction in the effort needed to generate the coarser-grained trace links, a very significant saving. 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 the trace links at the granularity of classes had 16% more false positives compared to the ones at the granularity of methods. This effect was much stronger on the granularity of packages which had over 40% more false positives

Table 3. Granularity-based tracing

Task	*Impact of lower granularity*
Acquisition	Correct and complete but less precise.
Utilization	Reduced quality but similar performance for short-term and long-term utilization.
Maintenance	Easier because of smaller quantity, completeness, and correctness.
Enhancement	Limited because can ignore code not identified on a coarser-grained trace.

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. However, it is the ability to provide complete and correct trace links at lower costs that is of particular interest in this chapter. Table 3 summarizes the benefits of granularity-based traceability for the four tasks of the trace life cycle.

Trace Maintenance heavily utilizes trace links. For example, if the change of a requirement requires a code change then the engineer uses trace links to identify the affected pieces of code. As a result of the change the requirement may then map to more or fewer elements of code (classes or methods). Trace maintenance ensures that the trace link is updated to reflect the new, correct, and complete relationship. Trace links on the granularity of Java classes identify all classes that need to be changed (i.e., because they are complete and correct). The lack of precision implies that the engineer must search through the methods of the identified classes but does not have to study the remaining classes not identified by the trace (no false negatives!). Consequently, the maintenance of coarse-grained trace links does not suffer from the problems identified in Section 3 where a significant portion of the source code had to be searched because of incompleteness. The following demonstrates the strong savings this entails.

In the ArgoUML case study we studied 38 requirements in detail. On average, a requirement traced to 247 methods and 46 classes. Clearly, these requirements were not trivial. It required roughly one-third of the effort to produce the coarser-grained requirements-to-classes traces as compared to the requirements-to-method traces. With that effort, the selective approach described in Section 3 only identifies 30% of the requirement-to-methods traces. With that level of effort, the remaining 70% (or 4,200) traces to methods would remain incomplete. While the requirements-to-classes traces identify all classes owned by a given requirement, the requirements-to-method traces would only identify some 30% of methods.

This leaves it up to the developer to guess the missing information. With the requirements-to-class traces, we know that only 46 classes (out of 645) are

affected in average by a requirements change. Since there is a 1:10 ratio of classes to methods in the ArgoUML system, an average of 460 methods have to be explored in more detail. The selective approach, on the other hand, guesses 30% of the methods correctly but misses out on the more than 4000 other methods that were not covered. It is much easier to refine 46 imprecise classes to methods than it is to discover 250 or so methods in a pool of over 4000 methods that were ignored because of incompleteness – this represents a 20-fold improvement in complexity and cost! This example illustrates that it is more beneficial to have completeness than precision for maintenance. The Siemens system and Video-on-demand system behaved similarly well.

Trace Enhancement benefits similarly from coarse-grained trace links. Recall from Section 3 that trace enhancement is necessary when an engineer discovers missing or incorrect trace links. This problem is obviously reduced or perhaps eliminated as the 3-fold reduction in effort increases the likelihood of correct and complete trace links at the granularity of classes. However, trace enhancement is still needed for upgrading the coarse-grained trace link to fine-grained trace links on demand. Thus, the problem has changed somewhat. Similar to trace maintenance, trace enhancement utilizes its own trace links. Only the classes identified by the trace link need to be refined but not all the other classes. Again, trace enhancement does not suffer from the problems of the selective strategy identified in Section 3 where a significant portion of the source code has to be searched because of incompleteness. The empirical data supporting this benefit is omitted because it is largely identical to the discussion under trace maintenance above.

Trace Utilization techniques differ with respect to handling false or missing trace links. This chapter does not provide a comprehensive overview here due to brevity. If the quality of the trace links is inadequate then trace enhancement is necessary and we already demonstrated that coarse-grained trace links outperform selective trace links during trace enhancement 20-fold. However, selective trace links perform better if the engineer is able to guess correctly which links will be needed. In our experience, this is rarely the case for long-term trace utilization. Only if trace links are generated for immediate use without later reuse, the selective approach may outperform our proposed strategy.

4.2 Value-Based Enhancements

Lessons learned:

- *Consider stakeholder value propositions to focus traceability for refinement.*
- *Use savings from lower granularity to focus trace refinement on high-value requirements.*

Thus far, we demonstrated that 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 an engineer may be willing to sacrifice some benefits to save cost, we believe that such a

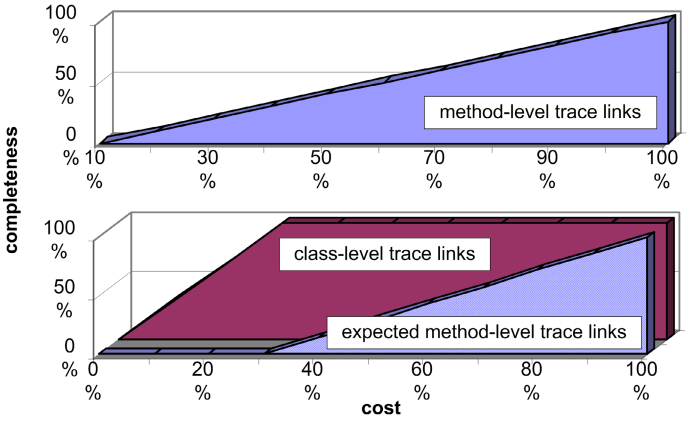

Fig. 2. Trace Acquisition with Selective (top) and Value-based Approach (bottom)

process must be guidable. In the following, we present a value-based extension to granularity-based trace acquisition, maintenance, and enhancement:

Value-based software engineering [3] 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 [3,5]. The definition of value depends largely on the domain, business context and company specifics. In essence, engineers can place value directly on trace links (i.e., this trace is important) or indirectly on the artifacts they bridge (i.e., this requirement is important and consequently its trace link to code as well). Our approach does not prescribe a particular value function. Indeed, the selective strategy discussed earlier could be considered a value-based approach, except, that the selective approach accepted incompleteness which we discourage in the value-based approach. The following demonstrates that not the initial acquisition but the enhancement should be value driven.

We discussed trace enhancement as a method for improving the precision of trace links in Section 3. Trace enhancement can be done at a later stage when an engineer discovers missing traces or incorrect traces. Trace enhancement can also be done early on to "upgrade" high-value traces as initially all traces including the high-value ones are only coarse grained.

Figure 2 (top) shows that the cost of trace acquisition for the selective approach maps directly to completeness. A 40% cost investment implies, in average, 40% completeness. Value-based trace acquisition follows a different pattern. We previously discussed that coarse-grained trace acquisition consumes in average 30% of the cost of fine-grained trace acquisition [13]. Figure 2 (bottom) depicts that with 40% of cost we reach 100% completeness on the class-level but have not yet created a single fine-grained trace link. The additional 10% funding can thus go into trace enhancement. Since trace enhancement is part of trace acquisition, we would expect it to be as effective as the selective approach but trace enhancement only has to refine those classes identified by coarser-grained trace links. As a consequence, if 100% of the cost is invested, we would expect the

value-based and selective approaches to be roughly equivalent. However, this expectation is wrong in two ways: (1) Few high-value traces result in most source code covered creating a negative effect; (2) Only overlapping trace links need to be enhanced for creating a positive effect. Both issues are discussed next.

Understanding the Diseconomy of Scale. Intuition says that if only half the trace links are important then only half the trace links need to be refined to a finer level of granularity – thus saving 50% of the cost. This intuition is misleading because 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.

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 3 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 and 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

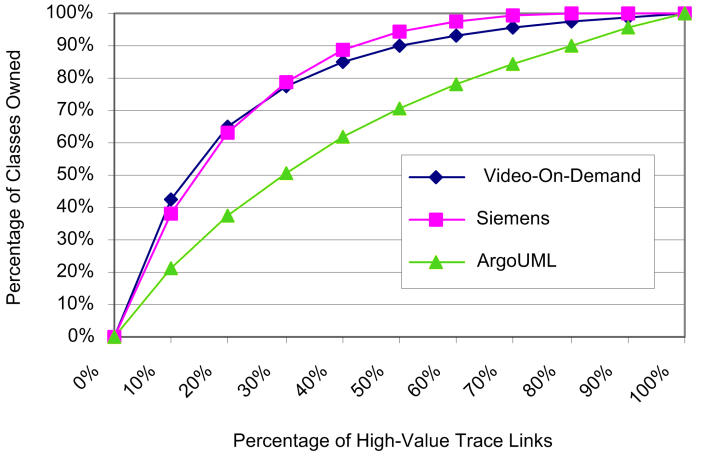

Fig. 3. Diseconomy by Enhancing Classes belonging to a High-Value Trace Link

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. This diseconomy is certainly another reason why the simple selective strategy discussed earlier was 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.

Enhancing Common Classes. Fortunately, there is also a positive effect that counters the diseconomy of scale discussed above. 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 in fact not necessary. Figure 4 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 [9].

The question is which of the classes in the various areas (overlapping or not) in Figure 4 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

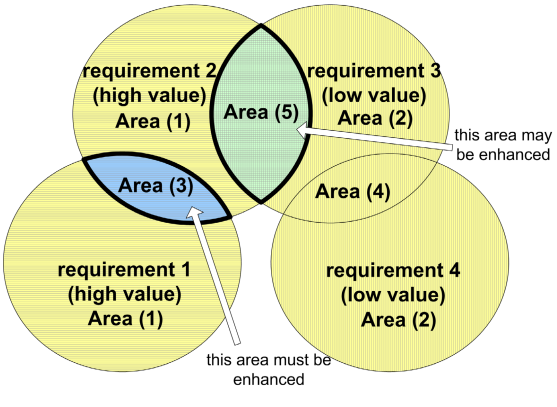

Fig. 4. Detailed Granularity is Only Necessary for Overlaps Involving Higher-Value Trace Links

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. 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 5 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 did not need to 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.

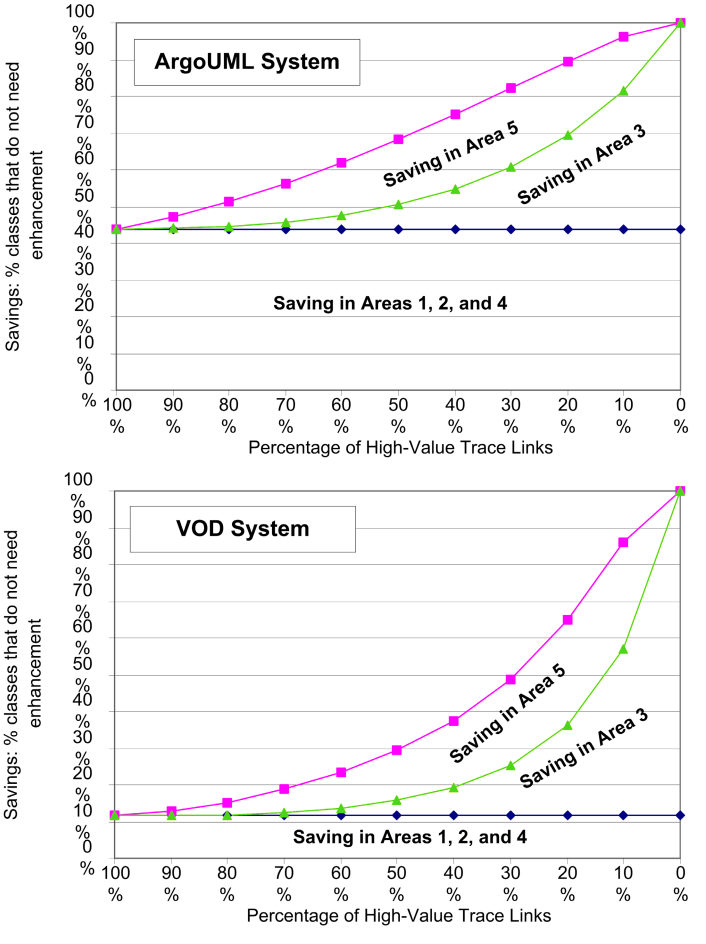

Fig. 5. Effort Saved due to Value Considerations in the Source Code (top: ArgoUML, bottom: VOD)

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. We will discuss later that 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 comes at no expense in terms of trace link quality.

So not only trace maintenance benefits from a 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 the 645 classes of the ArgoUML case study only 134 classes 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. We reduced the effort by 80%.

5 Discussion

It is well known that most software development effort goes into maintenance and evolution [5]. Consequently, more effort is spent on maintaining trace links or enhancing them than on acquiring them. This leads to a final lesson: *"Don't focus on saving effort during trace acquisition without considering trace maintenance and enhancement."*

However, while working and interacting with industrial partners (including Siemens Austria, Boeing Company, NASA, and geDV) we did observe legitimate reasons for wanting to limit trace acquisition efforts. Limited time and budget were obviously the more dominant reasons. The lessons we presented suggest investing traceability effort on what matters most and provide a way of limiting trace acquisition without seriously impeding trace maintenance and enhancement.

Granularity Trade-Off. Section 4 demonstrated that reducing the granularity of trace links reduces the complexity of trace acquisition by a factor of ten and cost by a factor of three. Coarse-grained trace acquisition produces correct and complete trace links (though imprecise ones) three times as fast. These links are thus more likely to be available early on. During trace maintenance and enhancement, they are mostly useful because of their ability to correctly identify where requirements do not trace to. We demonstrated that an average requirement change may affect dozens, even hundreds of methods. While this impact seems large, one must consider that this is only a small percentage of the total number of methods – the ArgoUML system defined almost 6,000 methods. While coarse-grained trace links may not identify the individual methods well (it errs by a factor of 2), it nevertheless identifies the much larger set of methods not needed to be looked at because if a requirement does not trace to a class then it also does not trace to any of its methods.

Value-based Trade-Off. If we would not have considered trace utilization, we could have stopped here. However, coarser-grained trace links are not as useful for trace utilization as finer-grained trace links. It is trace utilization that drives the quality needs of trace links. We thus demonstrated how to refine the granularity of trace links on a selective basis. This refinement process is guided by the engineer; however, we believe that it should follow value-based criteria. Interestingly, value-based trace acquisition does not appear scalable on a first glance. The problem is that classes and methods are often shared among multiple requirements. This causes a diseconomy of scale in that few high-value requirements, collectively, may own a larger share of the source code. Indeed, we found that 40% high-value requirements own between 50-80% of the source code. Under these circumstances, little is gained by following a value-based approach to trace acquisition. Fortunately, we also found that not every class traced to by a high-value trace link must be refined. In fact, only classes traced to by at least two high-value trace links needed to be refined. This resulted in a saving of 30-70% compared to a value-neutral approach. Of course, not even a value-based approach can guarantee accurate prediction of trace links that will be needed later. Trace enhancement, which is enabled by our strategy, improves the precision of a trace link that was incorrectly identified as a low-value trace link. It must be noted that our strategy does not identify high value trace link. This is done by the customer and/or engineer.

Tool support. The lessons presented do not prescribe a particular method or tool for doing trace acquisition, maintenance, and enhancement. The three strategies outlined in this chapter, brute-force, selective, and value-based, are applicable to any method or tool that is guidable (and most are guidable). However, trace acquisition is a mostly manual process and the value-based strategy does not require the engineer to change how to perform these manual tasks either. Rather, it guides them what traces should be done when and at what level of granularity but leaves it up to the engineer how this should be done.

Cost and Effort. We advocate that trace acquisition should always be done completely. The minimal investment of trace acquisition is the cost/effort needed to complete trace acquisition on a coarser level of granularity.

Correctness. A value-based strategy can significantly save cost and effort. This saving does not come at the expense of the quality among the higher-value trace links. All higher-value trace links are produced at the highest quality. Only low-value trace links are produced at a lower level of quality; but we have seen that even this quality reduction is moderate (between 15-30% more false positives depending on the case study).

Human Error. It must be noted that we ignored the issue of human error in this chapter. There is always some degree of error associated with trace links. The degree of error depends on a range of factors such as engineer's experiences, engineer's ability to recollect facts about the artifact and/or code to be traced or even tool errors. This chapter ignored these kinds of errors because it is not affected by our value-based strategy. Recall that this work does not prescribe a

Table 4. Requirements value based on typical ranges for business value feasibility

Value/Feasibility	Easy to realize (40% to 60%)	Hard to realize (40% to 60%)
high (30% to 80%)	High value (12% to 48%)	Low value (12% to 48%)
Low (20% to 70%)	Low value (12% to 48%)	Low value (12% to 48%)

different tool, method, or engineer for doing traces. This chapter only suggests what to do when.

Finding your value function. As pointed out above our strategy relies on the ability to identify the important trace links. We previously argued that typically between 15-50% of trace links are important (high value). This data is based on previous work on the importance of requirements during the software lifecycle. It is important to stress that arbitrary, user-definable utility functions can be used to experiment with different scenarios. The possible savings depend on the ability of the function to predict short-term and long-term utilization.

We found that the priorities of requirements represent a good proxy for stakeholder value. For software traceability, value-based software engineering means to produce better quality trace links for higher-value requirements or other valuable artifacts. It is common practice in industry to prioritize requirements or design elements according to their importance and feasibility ratings from success-critical stakeholders. Important requirements that are easy to realize have a higher value than unimportant requirements that are hard to realize. Table 4 indicates this relationship together with typical percentages for business value and ease of realization.

This utility function is just one example. However, we believe this function is reasonable in many contexts. Obviously utility functions can be optimized and calibrated to allow even higher savings. The dilemma is that it is not possible to devise a perfect utility general-purpose function that tells about short term and long term needs. Granularity allows us to save money in the short term and to significantly benefit the maintenance and later enhancement in the long run. A value-based approach to traceability is most likely to strike the right balance between the cost and benefits of traceability.

6 Conclusion

In this chapter we have presented lessons learned from three case studies. The lessons suggest a value-based approach to software traceability: 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. A value-based approach does not suggest ignoring trace links. On the contrary, this work strongly advocates the completeness and correctness of trace links. In fact, it is the goal of this work to accomplish completeness and correctness as quickly as possible, even at the expense of precision, and to then enhance (refine) the trace links as the budget allows.

We believe that 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. Our value-based strategy tells when to establish which traces but it does not tell how to do trace acquisition. It thus applies to any existing traceability method or tool that is guidable.

References

1. Abbattista, F., Lanubile, F., Mastelloni, G., Visaggio, G.: An experiment on the effect of design recording on impact analysis. In: Int. Conf. on Software Maintenance, pp. 253–259 (1994)
2. Bianchi, A., Visaggio, G., Fasolino, A.R.: An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models. In: Proc. of the 8th Int. Workshop on Program Comprehension, p. 149 (2000)
3. Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., Grünbacher, P.: Value-based Software Engineering. Springer, Heidelberg (2005)
4. Biffl, S., Heindl, M., Egyed, A., Grünbacher, P.: A Value-Based Approach for Understanding Cost-Benefit Trade-Offs During Automated Software Traceability. In: Proc. of the 3rd Int. Workshop on Traceability in Software Engineering, Long Beach, CA (2005)
5. Boehm, B.W., et al.: Software Cost Estimation with COCOMO II. Prentice Hall, Englewood Cliffs (2000)
6. Boehm, B.W., Huang, L.: How Much Software Quality Investment Is Enough: A Value-Based Approach. IEEE Software 23(5), 88–95 (2006)
7. Cleland-Huang, J., Zemont, G., Lukasik, W.: A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In: Proc. of the Int. Conf. on Requirements Engineering (RE), Kyoto, Japan, pp. 230–239 (2004)
8. Dohyung, K.: Java MPEG Player (1999), `http://peace.snu.ac.kr/dhkim/java/MPEG`
9. Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering (TSE) 29(2), 116–132 (2003)
10. Egyed, A., Grünbacher, P.: Identifying Requirements Conflicts and Cooperation. IEEE Software 21(6), 50–58 (2004)
11. Egyed, A., Heindl, M., Biffl, S., Grünbacher, P.: Determining the Cost-Quality Trade-off for Automated Software Traceability. In: Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), Long Beach, CA (2005)
12. Gotel, O.C.Z., Finkelstein, A.C.W.: An Analysis of the Requirements Traceability Problem. In: Proc. of the 1st Int. Conf. on Rqts Eng., pp. 94–101 (1994)
13. Heindl, M., Biffl, S.: A Process for Value-based Requirements Tracing – A Case Study on the Impact on Cost and Benefit. In: Proc. of the European Software Engineering Conf. and Foundations of Software Engineering (ESEC/FSE), Lisboa, Portugal (September 2005)
14. Huffman Hayes, J., Dekhtyar, A., Osborne, J.: Improving requirements tracing via information retrieval. In: Int. Conf. on Requirements Engineering (2003)
15. Lindvall, M., Sandahl, K.: Practical Implications of Traceability. Journal on Software – Practice and Experience (SPE) 26(10), 1161–1180 (1996)
16. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: bridging the gap between design and implementation. IEEE Transactions on Software Engineering 27(4), 364–380 (2001)

17. Neumüller, C., Grünbacher, P.: Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. In: 21st Int. Conference on Automated Software Engineering, Tokyo (September 2006)
18. Robins, J., et al.: ArgoUML, http://argouml.tigris.org/
19. Ramesh, B., Stubbs, L.C., Edwards, M.: Lessons learned from implementing requirements traceability. Crosstalk – Journal of Defense Software Engineering 8(4), 11–15 (1995)