Tracing All Around

Gerald Ebner [1] and Hermann Kaindl [2]

[1] Significant Software, Zeltgasse 14/11, A-1080 Vienna, Austria
ebner@significantsoftware.com
[2] Siemens AG Österreich, Geusaugasse 17, A-1030 Vienna, Austria
hermann.kaindl@siemens.at

Abstract. Many information systems are reengineered and redeveloped in practice, since they are legacy software. Typically, no requirements and design specifications exist and, therefore, also no traceability information. While especially the long-term utility of such information is well known, an important question in reengineering is whether installing it can have immediate benefits in the course of the reengineering effort. Are there even special benefits of traceability for reengineering?

In this paper, we argue for completely tracing all around from code over specifications to code in the course of reverse engineering an existing software system and its subsequent redesign and redevelopment. Experience from a real-world project indicates that it can indeed be useful in practice to provide traceability all around also for the developers and in terms of short-term benefits already during the development. We found several cases where traceability provided benefits that appear to be specific for reengineering. As a consequence, we recommend special emphasis on traceability during reengineering legacy software.

1 Introduction

Given a legacy information system, a usual task in software development is to come up with a new system that can substitute the old one. Most legacy software in practice does not have its requirements or design documented. Consequently, no traceability information is available either.

First of all, the requirements and the design of the old information system are to be reverse-engineered [1], in order to understand what the new software should be all about. Is it also useful to spend the extra effort for installing traceability information from the old implementation to the reverse-engineered design and to the requirements? What could be gained from that already in the course of the redevelopment? Shall traceability information be installed also between the requirements and the new design and the new implementation?

The thesis of this paper is that it is indeed useful to trace “all around”, i.e., to trace from existing code to design to requirements during reverse engineering, and from the modified requirements to the new design and the new code during development of the successor system. In effect, this means an integration of traceability during reverse

engineering with (more standard) traceability during forward engineering. In addition, we argue for having design rationale integrated, where the old and the new design decisions can be traced as well.

In support of this thesis, we present a case study from a real-world project of reengineering a legacy information system in the context of stock trading. The experience from this case study suggests that in the course of developing new software based on the code of an existing predecessor system, it provides benefits for the developers even in the short term to establish traces "all around". We found several examples of such benefits from traceability that appear to be specific for reengineering.

This paper is organized in the following manner. First, we present both our high-level view and our realization of traceability in reengineering legacy software. Then we show several cases of how it was useful in the course of a real-world case study. Finally, we discuss our approach more generally and relate it to previous work on traceability.

2 Traceability in Reengineering Legacy Software

Since for legacy software little or no documentation exists on the requirements, design and design rationale, usually also no traceability information is available. This makes reengineering such software systems particularly hard. Because reengineering efforts under those conditions are common-place in the software industry, it is important to investigate how to improve them, e.g., by installing traceability in the course of reengineering.

Unfortunately, there are severe time and resource constraints in real-world projects, and establishing traceability takes time and may be costly (unless it can be automated). So, from a practitioner's perspective an important question is, whether and how traceability can immediately help in the course of such a reengineering effort. The trade-off between cost and *short-term* benefits of traceability seems not to be well understood yet. We argue in favor of the usefulness of traceability in the course of both reverse engineering the old software and developing the new software, since we found empirical evidence for short-term benefits.

2.1 High-Level View

Our high-level view of traceability in reengineering is illustrated in Fig. 1 in a UML (Unified Modeling Language) class diagram that is enhanced by arrows.[1] Tracing all around in reengineering means to trace from the implementation of the old system to design and to requirements during reverse engineering, and from the modified requirements to the new design and the implementation of the new system during development of the successor system.

[1] For the standardized specification of UML see http://www.omg.org.

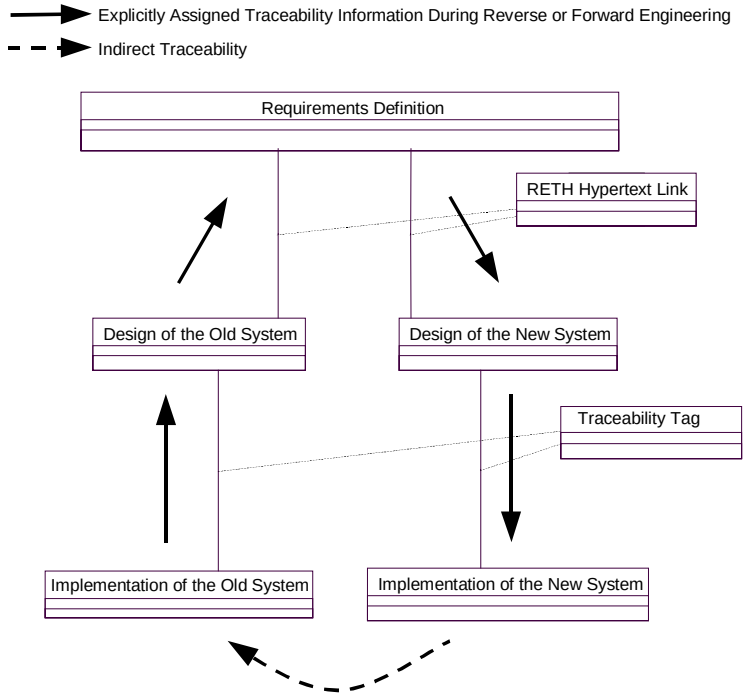

Fig. 1. An illustration of tracing all around

Once traceability is explicitly established from the old to the new implementation via design and requirements specifications, it is indirectly available between those implementations, as illustrated by the arrow with the broken line. From a theoretical perspective, these relations are included in the transitive closure of explicitly represented traceability relations. In practice, this means that no explicit relation between the implementations needs to be represented, since it can be derived.

In our view, there is only one definition of the requirements necessary, if the old and the new software are supposed to satisfy more or less the same requirements. Reverse engineering projects in practice will typically have to deal with at least minor changes or additions to the requirements [1]. Still, there can be a single requirements specification in the course of the reengineering effort, defining both the old and the new requirements.

2.2 Our Realization

In the case study reported below, the tool RETH (Requirements Engineering Through Hypertext) was used (for the RETH method and its supporting tool see, e.g., [7]). It was not only used there for capturing the requirements, but also the software design (both old and new) and traces. Hyperlinks in the RETH tool served as a means for installing traceability information in our realization, linking various artifacts in the

specification of the requirements with artifacts in the two software designs (also illustrated in Fig. 1). Installing hyperlinks in the RETH tool is inexpensive due to its semi-automatic support for link generation [9]. The mechanism for generating glossary links can also be utilized, e.g., for generating traceability links, based on textual references. An immediate advantage of such links is that they can be easily navigated for following traceability paths.

For technical reasons, however, the source code of the old and the new implementations had to be kept outside this tool. So, hyperlinks of that sort were infeasible to install, and we chose to use *traceability tags* in our realization of traceability among design artifacts and source code (see also Fig. 1). Installing traceability tags just means to insert a text string. For generating unique names of tags, we provided rudimentary tool support.

Fig. 2 illustrates in a more detailed UML class diagram the *metamodel* used, i.e., the model of how the models look like during reengineering and forward engineering. Since the traceability metamodels in the literature as discussed below are not sufficiently specific, we developed one ourselves. The top part (showing the object classes in white) is the previously published metamodel of RETH for requirements engineering [7, 8] (aggregated here by Requirements Definition). The other object classes (shown in grey) extend it for covering also design (shown lighter) and implementation artifacts (shown darker), including a simple representation of design rationale (Design Decision, which is part of Design). As indicated by Fig. 1 above, for reengineering those parts of the metamodel are instantiated twice: for the old and the new design and implementation.

Associations named Trace in the metamodel represent between which object classes traces are to be installed:

- between Requirements Definition and Design;[2]
- between several parts of Design and Implementation.

Especially for the latter, the granularity of traceability information is of practical importance. According to our experience, it should not be too coarse, e.g., to a source code module as a whole. But if it is too fine-grained, e.g., to each statement in the source code, then it becomes too expensive to install. As a balance, we chose to have a traceability tag assigned to each procedure of the source code as well as to each table definition, database trigger and stored procedure of the database script. This can be done with reasonable effort by the developers, and it can still be useful (see the experience from the case study reported below).

The representation of design rationale is also a practical compromise. Each Design Decision just describes the Design Chosen and the Alternatives Considered, since more elaborate structures like those in [12] would probably not have been maintained under the given time constraints of the given project. It was important, however, that the design decisions are traceable.

[2] In order not to clutter this diagram, we abstract here from the details of where exactly hyperlinks are installed.

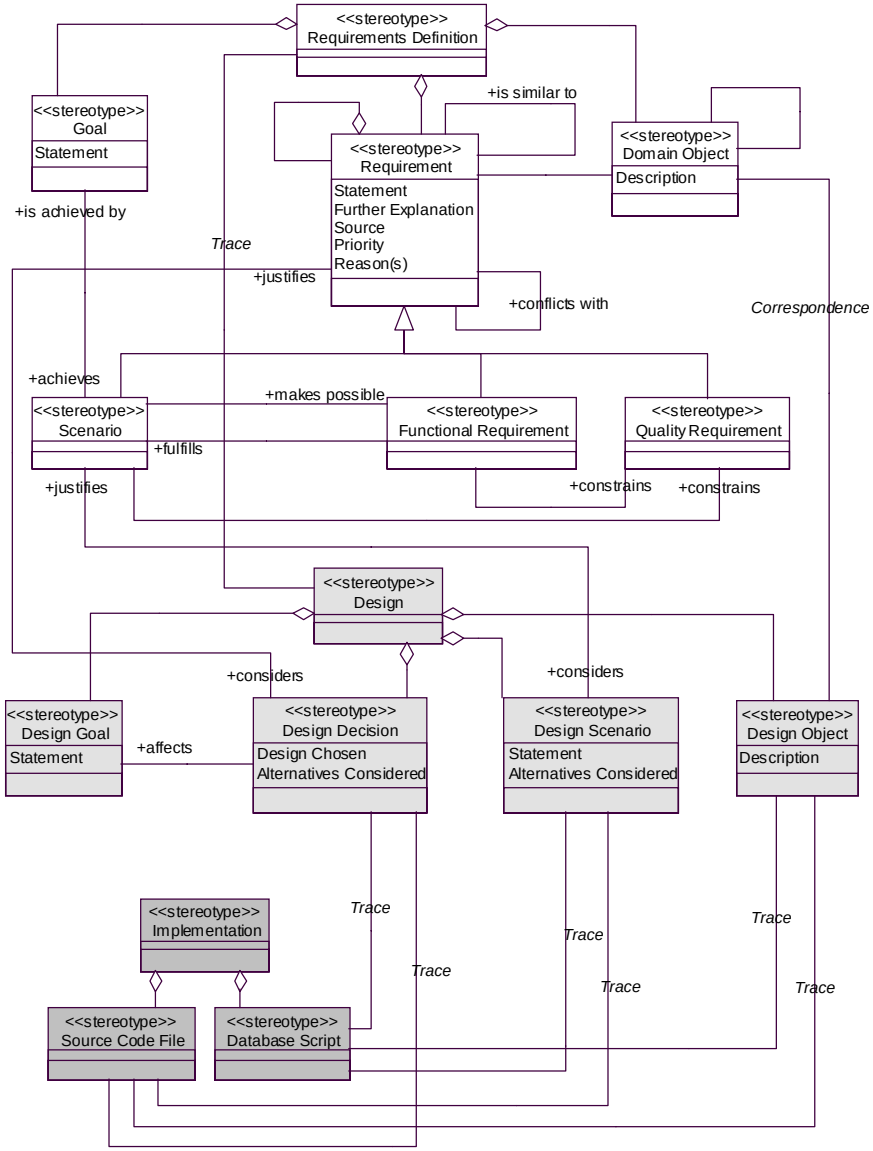

Fig. 2. The metamodel

3 A Case Study

The case study reported here was a real-world reengineering effort of legacy software in the context of stock trading. The installation and use of traceability happened in

"real-time" during the project. We describe the given task, the various roles in the case study and, in more detail, examples of concrete project experience. However, we focus exclusively on traceability and in particular its benefits for the developers of the new software.

3.1 The Given Task

More precisely, the given task was reengineering of the *Trade Bridge* software, which primarily makes continuous checks for consistency of received stock data and distributes those data to other software involved. Fig. 3 illustrates on an abstract level how the Trade Bridge software interfaces with those other software systems. The Front Office System delivers trade data, position data, etc. to Trade Bridge, where they are stored in a repository (implemented as a relational database). Trade Bridge checks those data for consistency and delivers the resulting data to the Mid Office System, the Risk Management System, the Ticket Printer and the Back Office System.

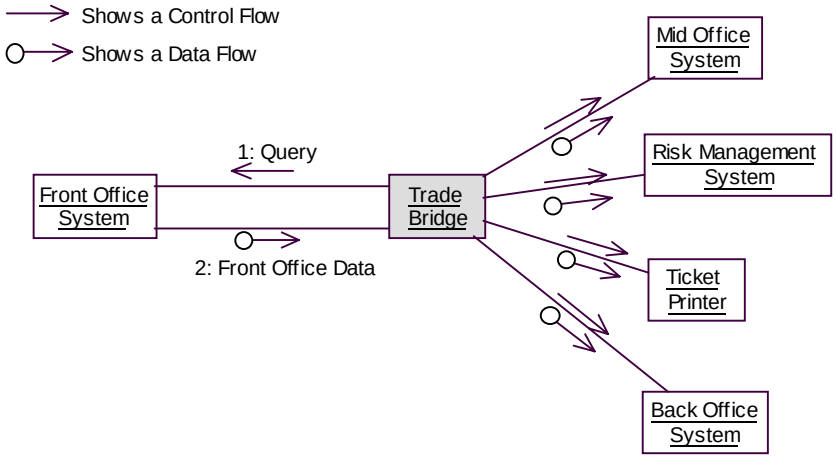

Fig. 3. Trade Bridge and its collaborations with other software

While for the other collaborations this short sketch should be sufficient in the context of this paper, we need to provide a few details of the collaboration between the Front Office System and Trade Bridge. As shown in the UML collaboration diagram in Fig. 3, the control flow happens from Trade Bridge to the Front Office System, while the data flow is the other way round. In fact, Trade Bridge is polling with queries for trade data, where both the queries and the data are transferred in ASCII text. The queries and the data transfer are asynchronous and, the response is on average in the order of minutes. It is also important to note, that transferring too much becomes a performance issue.

Fig. 4 illustrates the main part of the information model of the Trade Bridge software, i.e., an implementation-independent model of the data as viewed from outside the boundary of this software. The concrete data model is based on it, but optimized

for its implementation in a relational database, that serves as a repository. The most important part of this model for the purpose of understanding the case study is the association between Trade Data and Additional Trade Data, where the former reference the latter. The Front Office System delivers those data asynchronously.

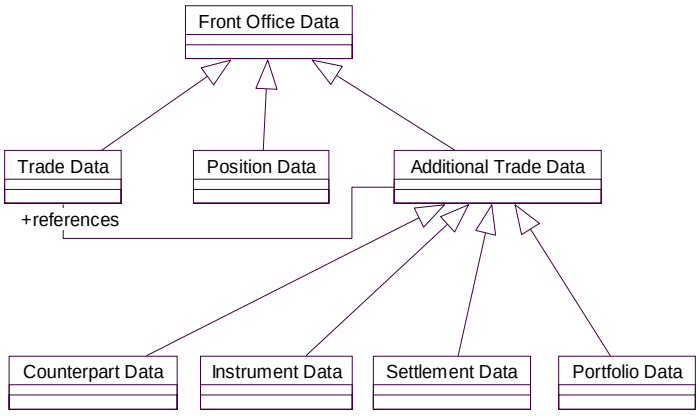

Fig. 4. Part of the Information Model of Trade Bridge

3.2 Various Roles in this Case Study

As usual in a software development project, several people participated in various roles. Since we describe our case study in terms of these roles, let us define them here:

- users:
  brokers and risk managers use the overall system, where Trade Bridge is an important part;
- developers of the old Trade Bridge software:
  they left the company and the project one month after its start;
- project leader for the reengineering effort:
  she was not involved in the development of the old Trade Bridge software;
- requirements engineer for the reengineering effort:
  the first author of this paper, who was also not involved originally;
- chief designer for the reengineering effort:
  same as the requirements engineer;
- developers of the new Trade Bridge software:
  including the first author of this paper; no single developer of the new software was involved in the development of the old Trade Bridge software.

3.3 Concrete Project Experience

In order to make the concrete project experience understandable, let us shortly sketch first the major approach taken in the course of reengineering Trade Bridge. Immediately after the start of the project, the requirements engineer acquired information about the requirements on the old software from its developers, rather than starting with design recovery based on the given implementation (and its source code). However, after the developers of the old software were not available anymore, everything (including requirements) had to be recovered.

The requirements engineer represented the requirements in the RETH tool and, in his role as the chief designer, he tried to figure out the connection of those requirements acquired from the developers with the implementation of the old software. He represented the resulting design information and the design rationale in the RETH tool (according to the metamodel illustrated above). The design rationale was partly acquired from the developers of the old software and partly hypothesized by the chief designer. He also installed traceability information immediately in the form of hypertext links and traceability tags, both during reverse engineering and forward engineering. In the following, we demonstrate immediate benefits of having this traceability information available through concrete examples.

Traceability in the Context of Additional Requirements
First we present an example, where traceability helped in the context of additional requirements. For the new Trade Bridge software, the users required additional functionality, which was captured, e.g., in the following requirement.

Functional Requirement *Synchronization of Referenced Counterpart Data:*

- Statement:

Trade Bridge shall transfer trade data to the Mid Office System only together with the corresponding counterpart data as delivered from the Front Office System, i.e., Trade Bridge must synchronize those data delivered asynchronously from the Front Office System.

As an immediate reaction to this additional requirement, the project leader thought about a rather straight-forward solution, to request all those data in one query that belong together. Instead of having to study this approach in detail, it was sufficient to consider the design rationale that was already reverse-engineered before with the help of the developers of the old software. In order to do so, however, it was necessary to locate this information.

First of all, the existing functionality related to the additional one was located easily in the hierarchically ordered requirements on the old software:

Automatically Transferring Trades
  Transferring Referenced Additional Trade Data

Then, through the traceability links in the form of hyperlinks, it was easy and efficient to locate the related design information and design rationale:

Design Decision *Determine which Additional Trade Data to Request*:

- Design Chosen

- Triggers that fire after the insert of new trades into the Trade Bridge database fill special Get tables with the key information of the given data.
- After all trades have been stored in the database, the Trade Reader subsystem asks the Get tables which additional trade data it shall request from the Front Office System (the number of tries is configurable) and processes these transfers.
- The number of tries in the appropriate record of the corresponding Get table is incremented.
- The count of additional trade data items that will be requested from the Front Office System at once is configurable.
- When additional trade data is actually inserted in the Trade Bridge database, triggers delete the corresponding entries in the Get table.

- Alternatives Considered

1. Request trade data and all associated additional trade data at once from the Front Office System by using a join query:
(+) No inconsistent data of the Front Office System is ever stored in the Trade Bridge database.
(-) Inefficient: A given instrument or portfolio is referenced by many trades so that the same data will be requested from the Front Office System many times.
(-) Unreliable: During the day time (when the Front Office System suffers from heavy work load) it is likely that huge join queries cannot be processed by the Front Office System.
2. Request all trade data and all additional trade data that was altered or inserted in the Front Office System during a given time frame. A prerequisite is that first all "old" front office data must be stored in the Trade Bridge database.
(+) A consistent state of the Trade Bridge database can be easily maintained without any effort of maintaining Get tables.
(-) Too much data would be transferred to the Trade Bridge system because a lot of the instrument and portfolio data of the Front Office System is not of interest.
(-) The performance of the resulting queries that have to be executed by the Front Office System would be very poor (in the current installation of the Front Office System TOO poor) because the information of the update time of each record is not indexed in the database of the Front Office System.

In fact, the alternative 1 already stored here corresponds to the design solution in question. Due to its main disadvantage given here as well (poor performance), it was easily and quickly rejected, based on this rationale previously acquired from the developers of the old Trade Bridge software.

Still, another solution had to be found for the new software. Rethinking the chosen design of the old software as given here led to a change and enhancement of the design for the new software, fitting the additional requirement:

Design Decision *Synchronizing Counterpart Data:*

- Design Chosen

- Triggers that fire after the insert of new trades into the Trade Bridge database fill a special Trade Sync Table with the ID of the given trade and the counterpart synchronization flag set to "N".
- The needed counterpart data is retrieved from the Front Office System via the Get mechanism.
- When counterpart data is actually inserted into the Trade Bridge database, triggers set the counterpart synchronization flag of the Trade Sync Table of all trades referencing the given counterpart data to "Y".
- Trade Bridge applications that need synchronized counterpart data must query the Counterpart Sync Flag of the Trade Sync Table before using the data of a certain trade.

While in the RETH tool there are many similar links, we show here just an example. The underlined string "Trade Sync Table" is in the RETH tool the source of a hyperlink from such a description to a referenced design object:

Design Object *Trade Sync Table:*

- Description:

The Trade Sync Table contains synchronization flags for all kinds of additional trade data (i.e., a counterpart, an instrument, a portfolio and a settlement synchronization flag) while only the first of the four is actually used. Flags may contain "Y" (synchronized) or "N" (not synchronized). The primary key of the table is the trade ID field and it is also a foreign key on the trades table.

Finally, it was also easy to locate the related parts of the implementation with the help of the traceability tags.

Traceability for Improving Completeness

The next example highlights the utility of traceability for improving completeness in the sense, whether the new source code covers everything covered by the old code. The criterion for coverage of the old code was defined in such a way, that traceability tags must be installed at least according to the granularity described above. That is, there must be a traceability tag in each procedure of the source code as well as in each table definition, database trigger and stored procedure of the database script (containing some 20,000 lines of code).

After the reverse engineering effort, developers of the new software made a code inspection which revealed, e.g., that out of hundreds of database triggers, slightly more than a dozen did not yet have traceability tags attached. One of those (for filling a table of instrument data) was particularly important, since its absence in the new software could have lead to recommendations of the risk management based on wrong data.

In more detail, the requirement originally reverse-engineered from the incomplete view of the code was the following:

Functional Requirement *Transferring Referenced Additional Trade Data:*

- Statement:

Trade Bridge shall request Additional Trade Data from the Front Office System when Trade Data was captured from the Front Office System which references portfolios, instruments, etc. which are not yet stored in the Trade Bridge repository.

The over-generalized assumption was that for all kinds of additional trade data the data transfer from the Front Office System to Trade Bridge can be reduced, whenever data are already in the repository of Trade Bridge. For instrument data, however, the corresponding check was commented out in the code of the old software. This was found in the course of the focused code inspections of those database triggers that had not yet a traceability tag assigned. The rationale for this implementation was recovered from risk managers: during the period since the instrument data were stored in the Trade Bridge repository, they may have changed!

After the traceability tags have helped to find that out, the wrong requirement was corrected so that it takes this exception into account. So, also the design of the new software and its implementation take it into account and, a major error in the new Trade Bridge software was avoided.

Traceability for Diagnosing Errors

As a final example, let us illustrate the usefulness of traceability for diagnosing errors. While the granularity of installed traceability tags was useful in the above example, it turned out to be insufficient in another case.

After the implementation of the new software, tests revealed a performance problem. The Front Office System interfacing with it was unable to deliver the amount of data requested by the new Trade Bridge software. Why does that not happen with the old Trade Bridge software?

The traceability installed all around made it easy to answer that question quickly, since it allowed finding immediately the corresponding parts in the old and the new implementation. In order to avoid this performance problem, the old software requested only five data records each per request, which is well hidden in the code of the following old procedure:

```
function MakeInstrumentQuery(): string;
// determine which instrument data to request from the
// Front Office System and build the “where” part of
// the query string °41°
begin
  ...
  SqlExecute('select distinct ID into strID from
  Instrument_Get where tries < 10 order by tries');
  while SqlFetch() and (nLines < 5) do begin
    strQuery = strQuery + strID + ', ';
    nLines = nLines + 1;
  end;
  if (nLines > 0) then
```

```
  ...
end;
```

The critical condition is "`and (nLines < 5)`". This is clearly below the granularity of the tags, where "`°41°`" is the traceability tag for this procedure.

Still, finding the correct diagnosis for this problem making use of the traceability information was possible within one hour, while it might otherwise have taken days. So, while the granularity of traceability information was insufficient to guarantee sufficient completeness, this information was still very helpful to solve a problem caused by an incomplete re-implementation.

4 Discussion

So, there is a trade-off in the granularity of traceability information. It relates to the more general trade-off between cost of installing and maintaining traceability information vs. benefits from having it available. Common wisdom suggests that traceability should pay off at least in the long run, including maintenance.

Unfortunately, the success or failure of a software development or reengineering project is not usually determined in practice from whether it provides, e.g., traces in order to facilitate maintenance later. The budget often just covers the cost to deliver a running system and, cost arising later from missing traces are outside the scope. This looks very short-sighted and it is regrettable from a higher perspective. However, many software systems developed are never deployed for several reasons. So, the focus is on delivering a running system on time so that it will be deployed, rather than on preparing for an uncertain future where other people will have to take care of the cost during maintenance, if at all.

That is why we argue for a distinction between immediate and long-term benefits. While we cannot provide quantitative data from our case study that would show already that traceability paid off within the reengineering effort, we observed and described cases where it provided benefits already in its course. In fact, we found also additional cases where traceability was useful in the short term. These were particularly related to changes in the requirements and the software design. In all those cases, the developers got more or less immediate reward for their effort of installing traces. Due to this reward, the motivation of the developers was increased to do this "extra" work, and this may finally result also in long-term benefits during maintenance.

5 Related Work

Within the last years, there was increasing focus on pre-requirements traceability and extended requirements traceability, i.e., on where the requirements come from in the first place [5, 6]. While the RETH tool can also support this aspect of traceability through hypermedia, it was not a major focus in the course of this case study.

Traceability among requirements, primarily from higher level to lower level requirements, is considered already for a long time and much better supported by tools [4, 10]. The tracing tool TOOR presented in [10] shares with our approach that it treats requirements and relations among them as objects and, that it can be applied to any other artifacts produced in a software project as well.

In [11], a requirements traceability metamodel is presented that also covers design and implementation. It is more comprehensive than ours presented here, since it includes also stakeholders, etc. However, our metamodel as illustrated in Fig. 2 is more concrete in specifying specialized meta-classes and their relationships for the various artifacts used to represent requirements, design and implementation. For the case study presented in [11], installing all those traces according to their metamodel was reported to be very expensive. They did not report, however, on short-term benefits or specific utility of traceability in reengineering.

There is much literature available on design rationale (see, e.g., [11, 12]), that presents quite elaborate argumentation structures. While such structures could be easily represented using hypertext in the RETH tool, according to our experience it would be too expensive to maintain them under the conditions of this real-world project. So, we found a practical compromise with the inclusion of a simple representation for design rationale and its integration with the overall traceability supported.

Recently, a special section in CACM (edited by Jarke) included several articles dedicated to more advanced proposals for traceability. In [3], it is argued for project-specific trace definitions and guidance of stakeholders in trace capture and usage (including enforcement). Our tool support for installing traces through the automatic link-generation facility of the RETH tool is a small but important step towards making life easier for those who are supposed to install traces in practice. In [2], it is argued for improving requirements traceability beyond the systems facet to the group collaboration facet and to the organizational facet of information systems.

So, this sketch of important work shows many aspects and a rather comprehensive view of traceability. Still, we could not find much in the literature about the trade-off between cost and utility of traceability, and especially not about short-term benefits. In current practice, however, it is very important to show the developers certain rewards to be gained immediately from installing traces. Also, we found no mention of specific utility of tracing during reverse engineering or even of tracing all around.

6 Conclusion

We observed special benefits of traceability for reengineering that we could not find in the literature. Partly, those benefits require tracing all around. The major lesson learned from several examples in our real-world case study is, however, that such traceability information can already help the developers (of the new system) in the course of reengineering legacy software. Such short-term benefits for the developers from applying this form of traceability can motivate them to provide traces that may even result in further long-term benefits later.

Acknowledgments

Mario Hailing and Vahan Harput provided useful comments to an earlier draft of this paper. Finally, we acknowledge the cooperation by the project members in the course of this reengineering effort.

References

1. E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.
2. G. De Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, J.W. Schmidt, C. Woo, and E. Yu. A three-faceted view of information systems. *Communications of the ACM*, 41(12):64–70, December 1998.
3. R. Dömges and K. Pohl. Adapting traceability environments to project-specific needs. *Communications of the ACM*, 41(12):54–62, December 1998.
4. R.F. Flynn and M. Dorfman. The automated requirements traceability system (ARTS): An experience of eight years. In R.H. Thayer and M. Dorfman, editors, *System and Software Requirements Engineering*, pages 423–438. IEEE Computer Society Press, 1990.
5. O. Gotel and A. Finkelstein. Extended requirements traceability: Results of an industrial case study. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering (RE'97)*, pages 169–178, Annapolis, MD, January 1997.
6. P. Haumer, P. Heymans, M. Jarke, and K. Pohl. Bridging the gap between past and future in RE: A scenario-based approach. In *Proceedings of the Fourth IEEE International Symposium on Requirements Engineering (RE'99)*, pages 66–73, Limerick, Ireland, June 1999.
7. H. Kaindl. A practical approach to combining requirements definition and object-oriented analysis. *Annals of Software Engineering*, 3:319–343, 1997.
8. H. Kaindl. Combining goals and functional requirements in a scenario-based design process. In *People and Computers XIII, Proc. Human Computer Interaction '98 (HCI '98)*, pages 101–121, Sheffield, UK, September 1998. Springer.
9. H. Kaindl, S. Kramer, and P.S.N. Diallo. Semiautomatic generation of glossary links: A practical solution. In *Proceedings of the Tenth ACM Conference on Hypertext and Hypermedia (Hypertext '99)*, pages 3–12, Darmstadt, Germany, February 1999.
10. F.A.C. Pinheiro and J.A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996.
11. B. Ramesh, C. Stubbs, T. Powers, and M. Edwards. Requirements traceability: Theory and practice. *Annals of Software Engineering*, 3:397–415, 1997.
12. B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, 1992.