Traceability in legacy software must often be reengineered, a process that involves both reverse and forward engineering. Typically, no requirements, design, or design rationale documentation for legacy software exist. This lack of documentation means an unavailability of traces as well, making reengineering difficult and expensive.

One way to improve reengineering efforts is to establish traceability—the ability to follow requirements information through the software life cycle—during the reengineering process. We propose tracing “all around”—from existing code to the legacy system's design to its 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 integrating traceability during reverse engineering with (more standard) traceability during forward engineering.

Our approach uses the Requirements Engineering Through Hypertext (RETH) method and tool to install traces during both reverse and forward engineering. Our experience in a real-world reengineering project suggests that all-around tracing can yield immediate benefits that appear to be specific to reengineering. In this article, we describe all-around tracing and present a case study to demonstrate how our approach can be useful in reengineering efforts.

A high-level view

To illustrate our high-level view of traceability in reengineering, we use a Unified Modeling Language (UML, see www.omg.org) class diagram enhanced with arrows, as shown in Figure 1. In all-around tracing for reengineering, we trace from the old system implementation to its design and requirements during reverse engineering, and from the modified requirements to the new design and system implementation during forward engineering.

Once we have explicitly established traceability from the old to the new implementation via design and requirements specifica-
tions, these implementations are indirectly connected. The arrow with the broken line in Figure 1 illustrates this relationship. From a theoretical perspective, the relationship is included in the transitive closure of explicitly represented traceability relations. In practice, this means that we do not need to represent an explicit relation between the implementations because we can derive it.

In our view, we need only one requirements definition if the old and the new software have more or less the same requirements. Although reengineering projects in practice typically deal with at least minor changes or additions to the requirements, a single specification defining both old and new requirements can be sufficient.

**Metamodel for all-around tracing**

Several traceability metamodels exist in the literature; however, they are not sufficiently specific for immediate application in practice. (See the sidebar, “Related Work on Traceability.”) The detailed UML class diagram in Figure 2 illustrates our metamodel for representing requirements, design, and implementation, as well as the connections between them. This is the model of how the models look during reverse and forward engineering in our approach.

We use RETH for requirements engineering and traceability. RETH is both a method and a tool that supports the method. The top section of Figure 2 (the yellow-green boxes, or metaclasses) is the RETH metamodel for requirements engineering. This part is aggregated in Figure 2 by requirements definition. The other classes extend this metamodel to include design (green boxes) and implementation artifacts (dark green boxes). Our extended metamodel also includes a simple representation of design rationale (design decision, which is part of design). Trace associations represent traces, which we install between requirements definition and design and between design and implementation.

For the latter traces in particular, trace granularity is of practical importance. According to our experience, traces should not be too coarse—for example, to a source code module as a whole. But if a trace is too fine-grained—for example, to each statement in the source code—installing it is too expensive. For balance, we put traceability tags at the procedure level. This requires reasonable effort and is still useful.

We include a correspondence association between domain object and design object to support additional traces. In our approach, a domain object is an abstraction of an entity in the problem domain. While we can clearly distinguish between domain objects and (software) design objects, a correspondence between them can certainly exist.

Our representation of design rationale is a practical compromise. Each design decision describes only the design chosen and the alternatives considered. It is unlikely that we could have maintained more elaborate structures within the project time constraints. It was important, however, that the design decisions be traceable.

During reengineering, we develop two models based on this metamodel: one during reverse and one during forward engineering. More precisely, a common requirements definition links the models of the old and the new designs and implementations for all-around tracing (as Figure 1 illustrates).

This approach is sufficient as long as the requirements are completely stable. We should anticipate, however, that requirements will change between the old and new systems. Figure 3 shows that the require-
Several articles in a special section (edited by M. Jarke) of the *Communications of the ACM* present high-level proposals for traceability.\(^1,2\) Still, we found little in the literature about the trade-off between the cost and utility of traceability, and in particular about its short-term benefits. In current practice, however, it is very important to show developers that rewards can be immediately gained from installing traces. Finally, we found no mention of the specific utility of tracing during reengineering or even of all-around tracing.

**Traceability tools**

Traceability among requirements, primarily from higher- to lower-level requirements, has long been considered and is well supported by tools. The Automated Requirements Traceability System (ARTS), for example, is a software database management system for requirements information management.\(^3,5\) Several tools for requirements management and traceability based on database systems are commercially available—for example, Requirements Traceability and Management (RTM, [http://www.chipware.com/](http://www.chipware.com/)), Dynamic Object-Oriented Requirements System (DOORS, [http://www.telelogic.com/](http://www.telelogic.com/)), and RequisitePro ([http://www.rational.com/](http://www.rational.com/)).

Of course, such tools could have managed traces in the Trade Bridge system as well as the RETH tool did. One of the main reasons developers chose RETH instead of one of these COTS tools was its automatic link-generation facility. Reflecting more recent research, Traceability of Object-Oriented Requirements (TOOR) treats requirements and their relations as objects similarly to our approach.\(^4\)

**Metamodels for traceability**

Balasubramaniam Ramesh and colleagues present a requirements traceability metamodel that also covers design and implementation.\(^5\) It is more comprehensive than ours because it includes, for example, stakeholders. It is rather high-level, however. For example, it simply states “design satisfies requirements,” instead of relating concrete design and requirements artifacts. Our metamodel defines in greater and more concrete detail specialized metaclasses and their relationships. In this sense, we think that our metamodel is easier to apply.

In their case study, Ramesh and colleagues report that installing traces according to their metamodel is very expensive,\(^5\) but they do not report short-term benefits or the specific utility of traceability in reengineering.

**Design decision support**

A great deal of literature on design rationale exists,\(^5,6\) most of it proposing quite elaborate argumentation structures. Balasubramaniam Ramesh and Vasant Dhar, for example, present a conceptual model for recording discussions and decisions about requirements in a network of issues, positions, and arguments connected by special links.\(^6\) We could easily represent such structures using hypertext in the RETH tool. They would have been too expensive to maintain under the Trade Bridge project conditions, however. As a practical compromise, we included a simple representation for design rationale and integrated it with the traces.

**Related Work on Traceability**

Hyperlinks in the RETH tool let us install traces by linking various artifacts in the requirements specification to artifacts in the two software designs (illustrated in Figure 1). The RETH tool’s semiautomatic support for link generation makes installing these hyperlinks inexpensive.\(^5\) The mechanism for generating glossary links can also semiautomatically generate traceability links based on textual references. An immediate advantage of such links is that users can easily navigate them to follow traceability paths. This navigation is possible in both directions because RETH hyperlinks are bidirectional.

For technical reasons, the source code of
the old and new software had to remain outside this tool, which made it infeasible to install RETH hyperlinks to or from source code. To compensate, we use traceability tags between design artifacts and source code (see Figure 1).

A traceability tag in our case study is simply a hexadecimal number surrounded by...

Figure 2. Metamodel for representing requirements, design, and implementation, and the connections between them. We install traces between requirements and design artifacts and design and implementation artifacts. A correspondence association supports additional traces between domain objects and design objects.

Figure 3. Requirements change. In our metamodel, the requirements definition allows for common requirements as well as requirements specific to the old or new system.
the special character ° (for example, °34F5°). Inserting the same tag at all the places in the various documents to be linked together can implement binary as well as higher-order relations, but a single tag by itself does not indicate traceability direction. From a practical viewpoint, this does not pose real problems because the metamodel clearly shows the direction.

We implemented a simple tool to generate and then find unique tag names. We can easily find tags in virtually any kind of document as long as the corresponding MS Windows tool provides full-text search. Further support for renaming, deleting, and checking tags might be desirable, but even the simple tool support available in our project was useful.

**A real-world case study**

Our case study involved the reengineering of Trade Bridge, a real-world legacy software system that continuously checks the consistency of received stock data and distributes that data to other software systems. The old Trade Bridge system was unreliable, difficult to maintain, and based on old technology. Because users wanted additional functionality, a new manager decided to reengineer the system. The project, from kick-off to full operation, lasted approximately one year.

During the project, developers installed and used traceability in real time. Here, we describe the task, the project’s basic reengineering approach, and some experiences, focusing exclusively on traceability and its benefits for developers of the new software.

Figure 4 illustrates on an abstract level how Trade Bridge interfaces with other software systems. The front-office system delivers trade, position, and other data to Trade Bridge, which stores the data in a repository (implemented as a relational database). After checking the data for consistency, Trade Bridge sends the checked data to the mid-office system, the risk-management system, the ticket printer, and the back-office system.

**Basic reengineering approach**

Immediately after the reengineering project began, the requirements engineer acquired information about the old software requirements from its developers. Fortunately, this was possible, and it saved some requirements and design recovery effort. Later, when the developers of the old software were no longer available, the reengineering team had to recover everything (including requirements). Our approach does not prescribe a process that requires either the original system developers or documentation. Rather, it defines a metamodel for representing useful information.

The requirements engineer/chief designer first represented the old software requirements in the RETH tool. In his design role, he tried to figure out how these requirements related to the implementation. He represented the resulting design information and the design rationale—which he partly acquired from the developers of the old system and partly hypothesized—in the RETH tool (compliant with the metamodel in Figure 2). During both reverse and forward engineering, he immediately installed traces as hypertext links and traceability tags. There were immediate benefits to having these traces available.

**Handling requirements change**

In the Trade Bridge project, traces facilitated the handling of requirements change with respect to related design decisions. One requirement, for example, concerned the accessibility of configuration parameters, which influences their storage method in
Trade Bridge. The chief designer reverse-engineered the design decision, “storing configuration parameters,” from the old Trade Bridge software code, with the design chosen, “configuration parameters are generally stored in ASCII files, or, more precisely, in MS Windows INI format.” Fortunately, the old system developers were still available for an interview, which revealed the functional requirement (see Table 1) that justifies this design. We represented this requirement, the related design decision, and the link between them in the RETH tool compliant with our meta-model. No other requirements or design aspects appeared to be related to this functional requirement.

Later, during forward engineering, users of the new software required different functionality. The requirements engineer formulated the new functional requirement, which Table 1 also shows. By this time, the developers of the old Trade Bridge software had left the company and the project. Nonetheless, the RETH tool traceability link meant that the old design decision was just a hyperlink away from the old requirement.

In this example, the old and the new requirements conflict. The new requirement therefore replaces the original, which the corresponding association in Figure 3 represents, thereby facilitating requirements change traceability. A new quality requirement related to the configuration parameters (see Table 1) also influenced the new design.

The requirements change made it impossible to keep the old design, which presumably only the replaced requirement justified. No other design aspect was related to the replaced requirement. Therefore, the new design decision (see Table 2) only considered the new requirements (see Table 1), and it was linked to them to facilitate traceability. As illustrated in Table 2, the design chosen was clearly preferable to the alternative design.

When requirements change, developers must determine the consequences of this change on the design. In reengineering,
Knowing which design decisions are related to replaced requirements is important. Developers should reconsider these decisions when designing the new system. Knowing the rationale behind old decisions can usually help them make better new decisions. In this way, traces installed during reverse engineering can assist developers during the forward-engineering process. This is one example of the short-term benefits of traceability.

In particular, this example shows how the following features of our traceability approach can be useful.

- We establish traces during reverse engineering (Figure 1).
- Requirements justify design decisions (Figure 2).
- New requirements replace old ones (Figure 3).

**Diagnosing reimplementation errors**

In the Trade Bridge project, traces also facilitated diagnosis of reimplementation errors. After implementation of the new Trade Bridge software, tests revealed a performance problem. The front-office system could not deliver the amount of data the new software requested—a problem that hadn’t occurred with the old Trade Bridge software.

During redesign and reimplementation, developers had used different names for artifacts and had distributed functionality differently. Finding the corresponding parts in the old and the new implementations could therefore be tedious and time consuming.

The installed traces, however, made it easy to find the corresponding parts in the old and new implementations immediately using our tool support for finding traceability tags, and thus helped determine the problem. The old software avoided the data-delivery problem by requesting only five data records at a time from the collaborating front-office system. This fact was well-hidden in the old procedure’s code, as Figure 5 shows.

The critical condition is \( \text{and} (n\text{Lines} < 5) \). This is clearly below tag granularity, where \( \#41\# \) is the procedure traceability tag. (In several cases not reported in this article, the granularity of installed traceability tags was sufficient.) Having the traces in place let us correctly diagnose the problem within one hour; with another method, such as searching artifact names, it might have taken days. Thus, traces can also help solve problems resulting from incomplete reimplementation.

In particular, this example shows how using our approach for all-around tracing in reengineering can be useful: establishing traces during reverse and forward engineering provides (derived) indirect traceability between the old and new implementations (see Figure 1).

**Lessons learned**

Our experience with the Trade Bridge reengineering effort made clear several benefits of implementing traceability in both reverse and forward engineering.

- Traces to the reverse-engineered requirements and design information can facilitate new software development within the same reengineering effort, if they are installed immediately.
- Although elaborate design rationale might be too costly to include, simple information on design decisions is feasible to maintain and trace in practice, even under heavy time constraints, making traceability even more useful.
- A trade-off in trace granularity exists. It relates to the more general trade-off between the cost of installing and maintaining traces and the benefit of having them available. Common wisdom suggests that traceability should pay off in

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Code from the old procedure. The old software requested only five data records at a time from the collaborating front-office system.}
\end{figure}
Unfortunately, whether a software development or reengineering project provides traces to facilitate later maintenance does not usually determine its success or failure in practice. Budget and time constraints are major issues in any project. Project budgets often cover only the cost of delivering a running system; costs arising later due to missing traces are outside their scope. From a higher perspective, this is both shortsighted and regrettable.

In addition, many software systems are developed but never deployed, so developers focus on delivering running systems on time. Preparing for an uncertain future is less important: other people will be responsible for any maintenance costs.

We therefore argue for a distinction between immediate and long-term benefits. Although we cannot provide quantitative data to show that traceability paid off within the Trade Bridge reengineering effort, it did provide benefits over the course of the project.

In fact, other cases in the Trade Bridge project also support traceability’s usefulness in the short term. These cases were particularly related to

- Additional requirements
- Software design changes
- Improved code completeness
- Iterative and incremental reverse engineering

In all cases, developers were immediately rewarded for their trace installation effort, which further motivated them to do this “extra” work. This situation is different from most development efforts with separate maintenance. Whenever a single team is responsible for all aspects of a reengineering project, the team can quickly harvest the fruit of “seeding” traces. Such short-term benefits can motivate them to provide traces that might also result in further long-term benefits later, possibly for other people than the developers.

Acknowledgments

We thank the project members involved in the Trade Bridge reengineering effort for their cooperation. Mario Hailing and Vahan Harput provided useful comments on an earlier draft of the article. Mark Sinclair made grammatical corrections. Finally, we acknowledge the very useful comments of the anonymous reviewers, who identified several weaknesses in an earlier version of this article.

References


For more information on this or any other computing topic, please visit our digital library at http://computer.org/publications/dlib.