A Systematic Approach to Software Evolution

Maritta Heisel
Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg
39106 Magdeburg, Germany
Tel. +49-391-67-12640
Fax +49-391-67-12810
email: heisel@cs.uni-magdeburg.de

Carsten von Schwichow
Fakultät für Informatik und Automatisierung
Technische Universität Ilmenau
98684 Ilmenau, Germany
Tel. +49-3677-69-4561
Fax +49-3677-69-4540
email: carsten.von.schwichow@tu-ilmenau.de

ABSTRACT
We present an approach to adjust existing software to new or changed requirements in an systematic way. The approach relies on a set of intermediate artifacts linked by mappings that bridge the gap between requirements and code. Those artifacts and the links between them can be constructed and maintained with reasonable effort. Additional support is supplied by bookkeeping and validation concepts. We demonstrate the usefulness of our approach by performing our method on a real-life application.

KEY WORDS
Reusability, Software Methodologies, Software Maintenance, Software Evolution

1 Introduction

Existing software engineering techniques usually treat the case where a *new* software system has to be built. All documents are developed from scratch, without any reference to existing documents. However, this situation is no longer realistic, because more often than not, no new software systems are constructed but existing systems are evolved and adapted to new requirements. Hence, a task that becomes increasingly important is to engineer *existing* software. Methods for software evolution – i.e., for adjusting existing software to new or changed requirements – are still missing, even though object orientation and component-based software engineering enhance the possibility to reuse existing software.

An important question for software evolution is the choice of an appropriate basis. Even though in the end software evolution leads to changing the code, it is not advisable to take the code as the starting point for evolution for the following reasons:

- The motivation for changing the existing system are additional or changed *requirements*. Hence, the requirements must be the starting point for software evolution.
- The different documents that make up the software system must be kept consistent. An evolution strategy that is based on code will almost certainly lead to neglecting the other documents. The result would be an undocumented and hence unmaintainable system.

We conclude that software evolution should be based on more abstract descriptions than only the code. Moreover, several descriptions on different levels of abstraction are necessary, because the gap between requirements and code is too large to be bridged by one mapping alone.

This paper proposes a number of artifacts to be constructed and linked by mappings, as shown in Figure 1. These artifacts can then be used to perform software evolution in a systematic way, following the different mappings, beginning at the requirements level and ending at the code level.

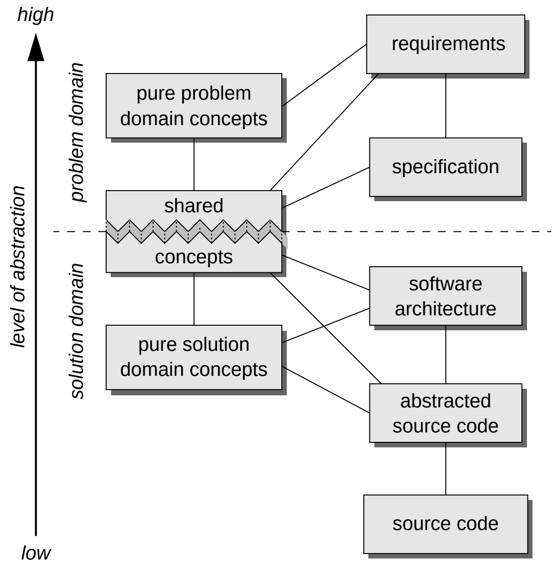

Figure 1. Artifacts and mappings

In the following, we describe the different intermediate artifacts (Section 2), the links between them (Section 3), and validation rules that allow us to check general integrity constraints (Section 4). A method for software evolution based on the different artifacts and mappings is presented in Section 5. We illustrate our approach by the real-life case study of a software system employed in German vocational schools in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Artifacts

We now explain the artifacts to be constructed (see Figure 1) in more detail. As advocated by Jackson and Zave [1, 2], we carefully distinguish a software development *problem* from its *solution*. The problem is expressed in the requirements. The requirements make statements about how the *environment* the software is intended to operate in will behave after the software is in function. They say nothing about the inner workings of the software. The solution of a software development problem, on the other hand, consists of the program code plus all documents set up when producing the code, including documentation.

In Figure 1, we use the term "concept". Concepts in the sense of this paper are (descriptions of) semantic units. That includes real world objects, relations, working principles, and operations. Solution domain concepts may be implementation language constructs, such as modules, object classes and functions, as well as architectural elements such as repositories, pipes, and filters, or even patterns and roles, like models, views, controllers, and observers.

2.1 Problem domain concepts

Problem domain concepts are those concepts that have a meaning within the problem domain. The problem domain comprises all parts of the environment of the software that it will have to influence or communicate with. That includes various (hardware) devices, but also other programs, such as server or client processes, for example operation system facilities like a graphical user environment. Note that for now "pure" problem domain concepts are not distinguished from "shared" (problem domain) concepts. The difference will be explained in Section 2.6.

Problem domain concepts can be found in the requirements, as the requirements will mention objects, relations and working principles that will be found or observed in the problem domain. Requirements state that some of the relations and operations will have to be performed or fulfilled by the software to be developed. Such properties are called *optative* properties [2]. On the other hand, there are properties of the problem domain which are always true, no matter if the software exists or what it does. Such fixed properties of the problem domain are called *indicative* properties and are said to form the *domain knowledge*. Note that the domain knowledge will never be expressed completely. This is because the domain knowledge on the one hand is typically very complex and on the other hand is taken for granted by those who use it every day. Nevertheless, it plays an important role in understanding the problem to be solved. Section 2.2 will underline the meaning of the domain knowledge for this approach.

2.2 Requirements and specification

As already mentioned, requirements make optative statements about the problem domain. According to Jackson and Zave [2], specifications are *implementable* requirements. This means that the specification must be expressed in terms of concepts accessible and manipulable by the software.

The specification is a description that suffices to build the software. It is derived from the requirements by using domain knowledge. For example, consider a library administration system to be required to set off an alarm whenever a book is removed from the library without permission. Unfortunately, the software is obviously not able to directly detect the removal of a book. There must be some technical facilities (sensors) that will send a signal to the software when a book that has not been checked out before is carried out of the library. Here the domain knowledge guarantees that this property will hold in the problem domain.[1] Without this knowledge, the requirement would not be implementable.

2.3 Code and abstracted code

A program's source code alone is not the most suitable document for understanding its working principles. Program comprehension becomes much easier when distracting details can be hidden, i.e., the code can be viewed on higher levels of abstraction. This can be done by identifying fragments of code for which a description of their purpose can be given. In the most simple case, this can be a descriptive name for the fragment. For example, a sequence of instructions which are found to exchange the values of variables *foo* and *bar* can be given the description "exchange *foo* with *bar*". This naming process can be applied recursively, so that if the surrounding instructions of the fragment mentioned above are found to sort the elements of an array *list*, this (bigger) code fragment can be given the name "sort *list*". The fragment "exchange *foo* with *bar*" thereby becomes a part of the fragment "sort *list*". Another application of code abstraction is to assign a short description to complex boolean expressions used as a branching condition. Such a step eases the understanding of what is being inspected by the condition and why the branching might be necessary. In Section 3, we show how this abstraction process can be performed with tool support.

The highest level of abstraction obtainable by this technique is the level of object classes, function groups, modules or alike. Abstractions of higher levels must be represented in the software's architecture.

2.4 Software architecture

Code abstraction is a good mechanism for raising the abstraction level of code in order to make it more understandable. But this technique is limited. As soon as a certain

[1] However, this "knowledge" may be falsified on a power failure, because then it might be possible to remove a book without the system recognizing it. Also, a skilled person might be able to remove the theft prevention mechanism from the book.

level is reached, code abstraction will not be useful anymore, because high-level interactions between a program's components are difficult, if not impossible, to be observed from within the code. Such high-level working principles must be expressed separately in the program's *architecture*. The architecture describes which components the program consists of and how they interact. That includes communication paths and protocols, as well as cooperation patterns and roles. The architecture is the highest-level description within the solution domain. If the architecture of an existing software system is not documented, it must be retrieved using design recovery techniques [3].

2.5 Solution domain concepts

Solution domain concepts can be found in the abstracted code and the software architecture. Examples are language constructs, such as modules, object classes, functions, and data structures, as well as architectural elements, such as components and connectors. Also, operations and relations can be concepts, like "exchanging the values of two variables" or "sorting an array". All of these concepts have a meaning within the solution domain. Once again, for now there is no distinction between "pure" solution domain concepts and "shared" concepts.

2.6 Shared concepts

In Sections 2.1 and 2.5 problem domain concepts and solution domain concepts have been introduced. In order to solve a problem, problem domain and solution domain must somehow be linked. Otherwise there would be no relation between the problem and the solution, and therefore it would be impossible to check if the solution really solves the problem. Hence, at least some of the problem domain concepts must be transferred to the solution domain, that is, they must have a proper representation within the solution domain. In other words, there must be a (non-empty) match between the concepts of the problem domain and those of the solution domain.

For example, in a library administration system, there exist books. On the one hand, these books exist in the problem domain as physical entities. On the other hand, they also exist in the solution domain, typically in the form of special data structures called records. Both concepts, the "physical book" (problem domain) and the "logical book" (solution domain) refer to the same semantic concept, namely the book. Therefore these two concepts constitute a single semantic unit called "book". Neither one of them will be useful without the other. If the system does not have any representations for the books, the library system will not solve the problem. If there are no physical books, the book records, just consisting of some bits, cannot be interpreted, and thus, will have no meaning in the problem domain.

As a consequence, only by merging problem domain concepts with matching solution domain concepts, shared concepts can be brought into existence. The key for this matching process are the semantics contained in the concepts. Only those concepts referring to the same semantic entity can be matched. While each of the original concepts remains in its domain, the unit formed by merging the two crosses the line between problem and solution domain. Only those concepts are said to be "shared concepts". All other concepts, for which no matching partner from the opposite domain can be found, are called "pure problem domain concepts" or "pure solution domain concepts" respectively.

3 Constructing mappings between artifacts

We now describe how to construct the mappings between the artifacts, as shown in Figure 1. Note that even tough the *construction* of the mappings may proceed in only one direction (e.g., the abstracted code is constructed from the code and not vice versa), *navigation* between the artifacts is always possible in both directions (e.g., for each abstracted code fragment, one can obtain the concrete code and vice versa). The mappings can be constructed and maintained with reasonable effort.

- Problem domain concepts are all the concepts used in the requirements or those introduced by domain knowledge.
- The mapping between requirements and specifications and the mapping of pure problem domain concepts to shared concepts have already been explained in Section 2.2. (There, we have mapped the pure problem domain concept of book removal to the shared concept of a sensor signal.)
- The software architecture represents the overall structure of the software in terms of components and connectors. It is a realization of the specification. Hence, each element of the architecture must be related to a shared concept. Moreover, for each pure solution domain concept, there must be an architectural element in which it is used.
- To construct the mapping between pure solution domain concepts and shared concepts, it must be documented for each pure solution domain concepts what shared concepts it supports. Since the software specification is expressed in terms of shared concepts, a pure solution domain concept is only useful if it is related to a shared concept.
- For each code fragment, we must be able to identify an architectural element it belongs to.
- The process of code abstraction as described in Section 2.3 can be performed using a tool that is currently under development. It allows the user select an arbitrary code section and assign it a name or description. The user will then be able to choose whether the code fragment should be displayed fully or by its

description only. Since this process can be applied recursively, the source code can be represented in the form of a tree, consisting of nested code fragments.[2] This kind of representation enables the user to choose the level of abstraction at which the code is to be displayed. It also maps higher level code structure and purpose to distinct pieces of code and thereby establishes links between abstracted code and ordinary code.

Note that we choose not to construct a direct mapping between the specification and the software architecture, because the structure of the two documents may be completely different. Hence, an indirect mapping via the shared concepts is easier to establish and to understand.

4 Validation rules

Constructing the artifacts and the mappings between them is a crucial prerequisite for our software evolution method to work. Therefore, we have conceived a number of validation rules that allow us to check general integrity constraints that the artifacts and mappings must fulfill, regardless of the particular problem and solution domains. Checking the validation rules can be performed automatically.

(1) Each solution domain concept must have an associated code fragment.

(2) Each architectural element must have an associated code fragment.

(3) Each concept mentioned in the requirements must be a shared concept itself or must have an associated shared concept.

(4) Each architectural element must have an associated shared concept.

(5) Each pure solution domain concept must have an associated shared concept.

(6) Each code fragment must have an associated solution domain concept.

(7) Specifications must not be directly associated with pure problem domain concepts.

In Section 5.2 we describe a mechanism that uses these validation rules to track changes down to the code level.

5 Evolution method

We now show how the intermediate artifacts and the mappings between them as displayed in Figure 1 can be used to adjust a so documented software system to new or changed requirements in a systematic way.

5.1 Descending from the requirements to the code level

Once the change requests are defined, we begin our software evolution process by adjusting the problem domain concepts. Changing or adding a requirement might influence the definitions of existing problem domain concepts, make some of them obsolete or add new ones. The set of possibly affected concepts can be obtained by following the existing links between requirements and problem domain concepts. Up to this point, no distinction between pure problem domain concepts and shared concepts is needed. However, every change in the requirements will yield changes in one or more shared concepts. Some of these influences may be direct, while others are indirect via associated pure problem domain concepts.

In any case, some shared concepts will have to be changed. This means that the *semantics* of the those shared concepts is subject to change. For example, additional attributes of an object may have to be considered, or relations must be represented that were not needed before. Whenever shared concepts are changed, this has an impact on solution domain concepts. Via the links to pure solution domain concepts and the architecture, the changes spread out to these artifacts, too. While small changes will influence a few solution domain concepts only, more complex ones are likely to have an impact on a larger number of concepts and might even make modifications to program's architecture necessary.

Finally, the changes have to be propagated to the code level. In order to do so, the code structure hierarchy must be followed to the actual code fragments. The abstracted code artifacts support this last part of the descent.

5.2 Using tags to guide the way

To support this descent, *tags* can be issued in order to guide the way to the affected code fragments. Every tag has a short piece of text associated with it and can be attached to an arbitrary artifact. There are two kinds of tags: a *change tag* is attached to each requirement, concept or architectural element which is altered, added or declared obsolete during the descent. Each change tag must contain a description of the necessary modifications. *Violation tags* are created automatically by checking the validation rules described in Section 4, which must hold at any time. Whenever a validation rule is violated, the responsible artifacts are tagged with a short description, indicating which rule has been violated.

During the descent the issued tags, especially the hints on them, can be used to determine which lower-level artifact must be considered for modification. For example, if a certain solution domain concept has an associated change

[2]In some cases, e.g. in aspect oriented programming, it might be desirable to link code fragments in a way that would violate the hierarchical structure. Such cases can be handled by bundling (semantically) related fragments into *groups*, which might be independent of the hierarchical code structure.

tag, all code fragments linked to this concept have to be considered for modification. The hints on the change tag now guide this process of seeking and modifying. Violation tags assist the process in the same way: if a violation tag states that a solution domain concept has no corresponding code fragment, this code fragment has to be added. Section 6.2 gives an example on the use of tags for tracking changes down to the code level.

6 Case study

In order to validate the method described above, a program called "LusdPlaner" was chosen as a case study. "LusdPlaner" is a Windows program used mainly by vocational schools to create schedules. It does not calculate the schedules but prevents collisions[3] as the user manually puts the lessons into the schedules. "LusdPlaner" is implemented in C++. At the time a request for a number of changes was made, it consisted of more than 20,000 lines of code, not including the framework that was being used. For reasons of space we will present the processing of only one of the new requirements.

The older version of "LusdPlaner" showed only one schedule in its application window. The users now requested the possibility to display another two schedules. These two schedules should appear beneath the main schedule, but should be much smaller and should not be used for editing. The purpose of the new "miniature schedules" should be that, whenever the user selects a lesson in the main schedule, the three schedules together would display the schedule of the respective teacher, student group and room.

6.1 Identifying and adjusting the concepts

First of all, the problem domain concepts used within the requirement had to be identified. The requirement mentioned the *schedule window*, the *selected lesson* and the *miniature schedule windows*, as well as *teachers*, *student groups* and *rooms*. Obviously, these expressions all have a certain meaning in the problem domain.

As the next step, the program's documentation and code was searched for matching representations of these concepts. Where documentation was not sufficient, reverse engineering techniques were applied to the code, in order to gain more abstract descriptions of it and make its meaning and purpose more comprehensible. In this process an object class "TScheduleWindow" was encountered, which was found to be a representation of the schedule window. Moreover, object classes "TRecTeacher", "TRecGroup" and "TRecRoom" were found, representing the relevant data of a teacher, a student group and a room, respectively. Therefore *teacher*, *group* and *room* turned out to be shared concepts.

[3]Whenever the user tries to assign a teacher, a student group or a room more than once at a time, this is called a collision. (However, there are certain cases where exceptions are possible.)

Finding a representation of the selected lesson was more difficult, though. It was clear that a lesson is an element of a schedule. It was found out that schedules were represented by a list of objects of the class "TRecBlock" which represented a consecutive block of identical lessons. When used in a schedule, a boolean variable "highlighted" was associated with each block, indicating whether the respective block was selected or not. In fact, the program did not allow the user to select a single lesson. Instead, only whole lesson blocks could be selected. Even more, if a lesson block was selected, all blocks of the same kind (same teacher, same student group, same subject) were selected as well. Although all selected lessons therefore referred to the same teacher and the same student group, the program allowed the user to assign a different room to each lesson block. On order to fulfill the requirements and make the selection of a *single* room schedule possible, the way of selecting lessons had to be changed, such that at any time there would be at most one selected lesson block. Nevertheless, it should be possible to *highlight* more than one lesson block, as could be done before. Only by this change was it possible to make the *selected lesson* a shared concept. Otherwise it would have remained a problem domain concept without a matching representation within the solution domain.

Of course there was no implementation of the miniature schedules in the original version of the program. In order to make them a shared concept, a proper representation within the solution domain had to be created. It turned out that it would be most reasonable to simply add a boolean member variable "isMiniView" to the already existing object class "TScheduleWindow", indicating whether the represented schedule should be displayed in normal or reduced size.

The requirement also states that miniature plans should not serve editing purposes. Therefore the behavior also had to be changed. The same boolean member variable "isMiniView" could have been used to solve this problem. But when other implications of having three instead of one schedule window were taken into account, it seemed more appropriate to separate control logic from view functionality, which was not the case for the "TScheduleWindow" object class. In fact, it was decided to use the model-view-controller (MVC) pattern, which was not regarded necessary before. This resulted in "TScheduleWindow" to be split into two classes, one containing the view functionality ("TScheduleView") and one containing the controller functionality ("TScheduleController"). This yielded a simple solution for having miniature schedules behave differently from standard schedules: while standard schedule objects are connected to a controller of class "TStandardViewController", miniature schedule windows are connected to controller objects of class "TMiniViewController", which do not permit modifications.

Table 1 shows the definitions of all concepts identified so far. Concepts having both a definition in the problem

concept	meaning in problem domain	meaning in solution domain
standard schedule window	interactive screen area showing schedule grid, normal size, editing possible	object of class “TScheduleView” having “isMiniView” = *false*, attached to a controller object of class “TStandardViewController”
miniature schedule window	interactive screen area showing schedule grid, reduced size, no editing	object of class “TScheduleView” having “isMiniView” = *true*, attached to a controller object of class “TMiniViewController”
teacher	person teaching at a school	record object of class “TRecTeacher”
student group	set of persons being taught at a school	record object of class “TRecGroup”
room	physical part of a building, enclosed by walls	record object of class “TRecRoom”
schedule MVC sub-system	-	combination of a schedule view, a schedule controller and a schedule model
schedule view	-	object of class “TScheduleView”
schedule controller	-	object of class “TScheduleViewController” or any of its sub-classes
schedule model	-	object of class “TSchedule”

Table 1. Definitions of all identified concepts

domain and the solution domain are shared concepts. Concepts having a definition in only one domain are specific to that domain. There are also pure problem domain concepts. For example, a teacher might be *ill*, in which case he or she cannot be assigned lessons.

6.2 Propagation of changes

As soon as the affected concepts were identified, the following steps had to be taken:

Adjusting the concept sets The three sets of *pure problem domain concepts*, *pure solution domain concepts* and *shared concepts* were adjusted according to Table 1. As a result, some links had to be removed, such as those related to the “TScheduleWindow” class, which had been discarded.

Adjusting the specification Since no explicit specification document existed nor was requested, this step was not performed.

Adjusting the architecture The MVC pattern was introduced into the software architecture. This resulted in additional links being established, such as the one between the pure solution domain concept “schedule view” and the respective component in the architecture document.

Identifying the affected code fragments In order to identify the affected code fragments, two kinds of tags were issued.

- **Change tags.** All concepts or architectural elements whose definition had been changed were tagged with a short hint specifying what had to be changed. Also, tags were added to artifacts which should be handled specifically. For example, the new “TScheduleView” class (*abstracted code*) was added a tag stating that it should be created by splitting the former “TScheduleWindow” class. Another tag was added, stating that a boolean member variable “isMiniView” should be added to the new class.
- **Violation tags.** All concepts, architectural elements and code fragments violating any validation rule were tagged with a hint specifying which validation rule was violated. For example, the newly added (pure solution domain) concept “schedule view” was tagged, because it had no associated code fragment, which violated validation rule (1). The associated architecture component was tagged for the same reason.

Changing the code fragments After the tagging process was performed, all directly or indirectly affected code fragments were changed, removed, or new code fragments were added, according to the respective mismatch. Each time a processing tag was handled, this tag was removed. For example, the “add boolean member variable *isMiniView*” tag was removed from the “TScheduleView” class, when this variable was actually introduced to the code and properly handled.

Checking the validation rules After all modifications on the code had been performed, the validation rules were checked again. All violation tags were removed if the formerly violated rule was now met. Artifacts remaining tagged were examined (and modified) again, until all tags could be removed.

7 Related work

Our systematic approach to software evolution makes use of different established techniques of software engineering.

The conceptual basis – in particular the distinction between problems and solutions and requirements and specifications – is taken from the work of Jackson and Zave [2]. This conceptual basis allowed us to identify and distinguish the different artifacts of Figure 1.

Our approach allows us to trace requirements from the requirements document to all the other artifacts produced, including the code, and vice versa. We use requirements tracing as a means to locate those parts of the code affected by new or changed requirements. In addition to locating the affected code, our method also provides guidance for really performing the changes.

In the literature, requirements tracing is primarily considered to be a technique to support requirements engineering. As Jarke [4] puts it, "(A trace) must capture linkages between the documents produced during a *requirements process*." (emphasis ours). Tools (e.g., [5]) are used to keep requirements documents up-to-date and to *document* their relation to other artifacts, but not to *change* the software. Hence, requirements tracing is part of our approach, but does not cover it entirely.

Reverse engineering [6] and design recovery techniques [3] are useful when constructing the different artifacts.

Before incorporating changes, it might be appropriate to refactor the code, i.e., to enhance the code structure without changing its semantics [7].

Lightweight approaches to software development, for example extreme programming [8], construct software in an incremental way. However, the increments are performed in a relatively short period of time. Since the lightweight approaches do not require much documentation, they cannot support the evolution of a software system over a long time period very well, as is the case with our method.

Let us note that all the work mentioned in this section only covers isolated aspects of our method, the added value of which consists in combining different notions and techniques to conceive a concrete process for software evolution.

8 Conclusions

In this paper, we have presented a systematic approach to software evolution, relying on a number of different artifacts that bridge the gap between requirements and code, and mappings between the artifacts. We have shown that the documentation constructed in this way is suitable to perform changes of an existing software system in a systematic way.

In particular, the contributions of this work are the following:

- We have identified a set of artifacts on different levels of abstraction that makes up a suitable basis for software evolution. The artifacts reflect the distinction between problem and solution domains.
- We have shown how to construct mappings between the different artifacts that can be used to navigate among the artifacts, allowing us to identify those parts of the code that must be changed in order to adjust the software system to new or changed requirements.
- We have identified validation rules that help avoiding errors in the construction of the artifacts and the mappings.
- We have given a concrete method for incorporating changes in a software system. Besides the artifacts and mappings, we have introduced bookkeeping facilities that support software developers in actually performing our method.
- We have pointed out possibilities for automation and tool support.
- We have shown the feasibility and the appropriateness of the method by a real-life application.

In the future, we intend to provide tool support for the whole method, i.e., for constructing and maintaining artifacts and links, as well as for performing evolutionary changes.

Acknowledgment We thank Thomas Santen for his comments on this paper.

References

[1] Michael Jackson. The world and the machine. In *Proceedings: 17th International Conference on Software Engineering*, pages 283–292. IEEE Computer Society Press / ACM Press, 1995.

[2] P. Zave and M. Jackson. Four dark corners for requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. Also availble under http://www.research.att.com/˜pamela/ori.html#fre.

[3] Ted J. Biggerstaff. Design recovery for maintenance and reuse. In Robert S. Arnold, editor, *Software Reengineering*, pages 520–533. IEEE Computer Society Press, 1992.

[4] Matthias Jarke. Requirements tracing. *Communications of the ACM*, 41(12):32–36, December 1998.

[5] Francisco A. C. Pinheiro and Joseph A. Goguen. An object oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, March 1996.

[6] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: A roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.