TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DISSERTATION

Managing Dependencies in Complex Global Software Development Projects

Ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften/der Naturwissenschaften unter der Leitung von

Prof. Stefan Biffl
188 (Institutsnummer)
Institut für Softwaretechnik und interaktive Systeme

Eingereicht and der Technischen Universität Wien
Fakultät für Informatik
von

Dipl.Ing. Matthias Heindl
9826997
1230 Wien, Maurer Langegasse 39-41/1/6
Matthias.Heindl@qse.ifs.tuwien.ac.at

Wien, am

eigenhändige Unterschrift

DEUTSCHE KURZFASSUNG

Global verteilte Softwareentwicklungsprojekte (GSD Projekte) sind komplex aufgrund zahlreicher Faktoren, wie z.B. einer üblicherweise hohen Anzahl von Anforderungen, der Verteiltheit der Projektteilnehmer, und der Unmenge an Abhängigkeiten in Arbeitsergebnissen (Anforderungsbeschreibungen, Testfallspezifikationen, Source Code).

Eine wesentliche Herausforderung in solchen Projekten besteht darin, all diese Arbeitsergebnisse konsistent zu halten. Dies ist wichtig, um sicher zu stellen, dass die Dokumentation durchgängig und aussagekräftig bleibt und um Fehler in der Software aufgrund von fehlerhafter Dokumentation zu vermeiden. Die Konsistenz zu erhalten ist sehr schwierig wenn Änderungen nur punktuell in Dokumenten vorgenommen werden und nicht auch in anderen verwandten Dokumenten nachgezogen werden. Das Erfassen und Verwalten von Abhängigkeiten zwischen Arbeitsergebnissen (Dependency Management) ist entscheidend, Konsistenz in den Arbeitsergebnissen zu wahren. Allerdings haben bestehende Dependency-Management-Ansätze folgende Schwachstellen:

- Sie beinhalten keinen Planungsschritt, der dabei unterstützt abzuschätzen, wie viel Aufwand für das Dependency Management (DM) im Projekt notwendig sein wird. Das führt dann oft zu unsystematischem, schwer nachvollziehbarem DM, das noch dazu mit Mehrkosten verbunden ist.
- Sie behandeln alle existierenden Abhängigkeiten als gleich wichtig; die Unterscheidung in wichtige, wertvolle Abhängigkeiten und solche, die nicht wertvoll sind, wird nicht in nachvollziehbarer Weise gemacht.
- Das Erfassen von Abhängigkeiten ist teuer und eine fehlerbehaftete Tätigkeit. Das Erfassen von Abhängigkeiten zwischen Arbeitsergebnissen, die in unterschiedlichen Tools verwaltet werden, wird nicht ausreichend unterstützt (aufgrund mangelhafter Toolintegrationen).

Moderne high-quality Ansätze für DM sollen folgende Kriterien erfüllen: sie sollen (a) einen entsprechenden Planungsschritt beinhalten, (b) Abhängigkeiten entsprechend ihres Wertes behandeln (um Aufwände für das Verwalten von unwichtigen Abhängigkeiten zu vermeiden), und (c) einfaches und billiges Erfassen und Warten von Abhängigkeiten auch über Toolgrenzen hinweg unterstützen. In dieser Arbeit behandle ich die folgenden Forschungsthemen, um die genannten Kriterien zu erreichen:

1) *Planung des Dependency Managements*: Diese Arbeit beinhaltet ein "Tracing Activity Framework (TAF), das einen Projektleiter dabei unterstützt, unterschiedliche DM Ansätze für sein Projekt vorab zu berechnen, um Aufschlüsse über anfallende Aufwände für das DM zu bekommen und den günstigsten Ansatz zu wählen.

2) *Erfassung und Verwaltung von Abhängigkeiten*: Diese Arbeit beinhaltet einen wert-orientieren Ansatz für DM, um auf high-value Abhängigkeiten fokussieren zu können. Ausserdem beinhaltet sie eine prototypische Implementierung eines integrierten DM Ansatzes, der DM über Toolgrenzen hinweg unterstützt (zwischen einem Anforderungsmanagement-Tool und einer Entwicklungsumgebung).

3) *Anwendung von erfassten Abhängigkeiten*: Diese Arbeit beinhaltet ein Konzept für ein auf erfassten Abhängigkeiten basierendes Notifikationssystem, das die zielgerichtete, zeitgerechte Kommunikation zwischen verteilten Projektteilnehmern unterstützt.

Diese Beiträge sollen das Kosten-Nutzen-Verhältnis von DM Ansätzen verbessern. Zur Evaluierung meiner Ansätze habe ich empirische Studien im industriellen Kontext gewählt.

TABLE OF CONTENTS

ABSTRACT

Global software development (GSD) projects are complex due to the high number of requirements, global distribution of project participants, and high number of dependencies between artefacts (e.g., relationships between requirements, or between requirements and test cases). A key challenge in GSD projects is to cope with requirement and artifact changes that occur concurrently along the life cycle. Often changes done only punctiform in certain artefacts threaten the consistency among artefacts. Managing dependencies is crucial for implementing changes consistently. Current approaches for dependency management, e.g. manual requirements tracing approaches or automated trace generation approaches, have the following shortcomings:

- These approaches focus mainly on how dependencies can be captured (e.g., requirements tracing) and they lack a planning step that (a) defines which dependencies should be captured by which tools and when, and (b) allows to compare dependency management alternatives in terms of expected effort and quality. This leads to unplanned, unsystematic (ad hoc) dependency management that often overruns the available budget and makes it hard to reproduce why some traces are captured and others not, and make keeping the overview in a complex GSD project much harder.
- Current approaches for capturing dependencies, e.g., requirements tracing, treat all requirements and their dependencies as equally important. The determination which dependencies are important and valuable is not supported by these approaches.
- Capturing dependencies is still an expensive and error-prone activity despite of the approaches reported in practice. Especially traceability across tool borders is an open issue (due to tool integration limitations).

Up-to-date hiqh-quality approaches for dependency management should meet the following criteria: (a) availability of dependency management planning support, (b) management of dependencies according to their value, and (c) cheap (feasible) capture of dependencies (also across tool borders).

In this work I provide research contributions in the following areas:

1) *Planning dependency management*: This work provides a tracing activity framework (TAF) that supports a project manager in (a) defining which dependencies should be managed and (b) comparing dependency management approaches in advance (before dependency management starts in the project) to find the most cost-effective approach.

2) *Capturing dependencies explicitly, systematically, and consistently*: This work provides a value-based requirements tracing approach to focus dependency management efforts on high-value dependencies. Furthermore, I provide a prototype implementation of a tool integration that supports capturing dependencies across tool borders (between a requirements management tool and a development environment).

3) *Application of explicit dependencies (traces)*: This work provides a concept for a dependency-based notification system that supports communication and in-time notification of GSD project team members.

These research contributions significantly improve the cost-benefit of available dependency management approaches. As evaluation concept for these contributions I used empirical case studies and report their results.

Acknowledgements

I want to thank Professor Stefan Biffl for guiding me through my PhD and for spending lots of time for providing very constructive feedback for my research topics. Furthermore, I thank Paul Grünbacher and Alex Egyed for collaboration in particular areas, and my colleagues Dietmar Winkler, Barbara Schuhmacher, Dindin Wahudin, and the participants of the REMESE workshops for interesting discussions and experience exchange. I also thank the diploma students who contributed to this work, especially Gerhard Totz and Christoph Radl.

Furthermore, I want to thank my colleagues at Siemens IT Solutions and Services: especially my boss Franz Reinisch for heavy but fruitful discussions and for giving me the freedom to work on phd-related topics; I have to thank him and Josef Withalm for offering me the possibility to combine research and practical work in the Support Center Configuration Management. I also want to thank Kurt Hofmann and Walter Wölfel, and Peter Brunner, Georg Mayer, and Manfred Proksch for supporting me during my work.

Last but not least, I want to thank my parents for supporting me in my studies.

List of tables

List of figures

1 INTRODUCTION

Software development is about producing computer programs. These computer programs provide functionality that fulfills particular user needs. Computer programs can be self-contained products or integrated into a system. These software-intensive systems also contain non-software components, e.g. a train communication and coordination system consists of hardware (trains, switch plates, cabling) and software (providing communication and monitoring functionality to see the status of trains and communicate with the train drivers).

Software companies usually setup software development projects to build software. These projects have a defined start and end date, a given budget, and a number of project participants that work together. The result of such a project is the software (computer program) ready for usage.

In the early ages of software development, project participants shared the same office rooms (or rooms which are close to each other) to foster communication and collaboration. Software companies still perform such colocated projects, but recently a new type of project has gained in importance: the globally distributed software development project which is also referred to as Global Software Development.

Particularly medium to large software development companies follow this trend towards global software development (GSD), where software development teams are not longer colocated (in one office) but distributed around the world, e.g., there may be a development team in Romania, the test team in China, and project management located in Austria. Such globally distributed projects emerged over the last years due to reasons like cost advantages or market proximity [67].

Besides these economic benefits global software development projects make software development more complex than "normal" colocated projects.

1.1 Complexity of (Global) Software Development Projects

The term "complex" can be generally defined as "*composed by two or more parts, hard to separate, analyze, or solve*" (Webster dictionary). A complex system can be defined as "*a whole made up of complicated or interrelated parts*" or "*a complex system is a system whose properties are not fully explained by an understanding of its component parts. Complex systems consist of a large number of mutually interacting and interwoven parts, entities or agents*" (Webster dictionary).

Even "normal" collocated projects with a certain size can get very complex due to the following factors:

- *Number of participants*: the more project participants exist in a project, the more complex is the project. The increased complexity stems from the increased challenge for project managers to steer the project, coordinate project participants and their tasks, and monitor the project status.
- *Number of requirements:* The higher the number of requirements, the harder is it to manage them and to keep the overview on dependencies between requirements. Project participants often use requirements management tools in projects with a high number of requirements in order to manage the requirements and their dependencies. Requirements management tools provide different filters and views and therefore support project participants to keep the overview on requirements.
- *Number of artefacts:* The bigger the size of the desired software, the more artefacts exist in the project. The term "artifact" summarizes all kinds of work products and elements of work products that evolve during the project in addition to requirements, e.g. design elements

(components), pieces of source code (source code classes and or methods), test cases, and additional documents. The more functionality is expected from the software product to be built, the more design elements, pieces of source code, and test cases will probably exist in the project. The result is increased project complexity.

- *Number of dependencies* between requirements and artefacts: requirements and artefacts do not exist in isolation from each other. Instead, there are dependencies between them which make the complexity (a large number of mutually interacting and interwoven parts, entities or agents). For example, design elements represent one or more requirements in the architecture model of the desired software, i.e. there is some relationship (or dependency) between them, e.g. if you change one requirement you would also have to check the design elements to keep requirements and architecture consistent. This in turn means that a change of a particular requirement or artifact has consequences on other requirements and artefacts (like in fishing nets where a move of one node makes some other nodes also move). Software development projects - where multiple project participants work on different artefacts - are complex, because one change in one artifact might affect other artefacts.
- *Volatility of requirements and artefacts*: the complexity of multiple project participants working on several interdependant artefacts in a software development project is increased by the volatility of artefacts. Volatility refers to the frequency of change of an artifact. The more often requirements or artefacts change, the more often other artefacts have to be checked for consistency.

In addition to these challenges global software development projects bring along further issues that increase the complexity of a software development project:

- *Distribution of project participants:* high distances between project team members that impose considerable costs on meetings
- *Time zone differences* that restrict the availability of project team members at other sites;
- *Fragmented project information and communication*: Usually, there should be some kind of project repository containing all data from whatever site of the project. In reality each site has its own repositories (or way of storing its intermediate products) and not all data is collected in one central repository. That means that some changes to artefacts in one repository are not immediately visible at other sites. In cases where these changes are relevant for other sites (because of dependencies that exist between artefacts developed at different sites), the results are inconsistencies or delayed adaptations at those sites. For example, changing requirements (at site A) have consequences on design elements (site B) and source code (site C, D).
- *No or limited project management overlook*: results from the fragmented project information.

In a nutshell, the complexity of software development projects results from multiple project participants working in parallel at different sites, creating and maintaining artefacts that have dependencies among each other. Managing these dependencies in complex software development projects is the key to cope with project complexity, as explained in the following subsection.

1.2 Dependencies in Global Software Development Projects

Software development projects usually follow given software development processes to build the desired software. A software development process describes a variety of tasks or activities and their sequence to build a software product, e.g. the Rational Unified Process [114], or Siemens Software Engineering Method (stdSEM) [122]. These processes also prescribe artefacts to be created during the project, e.g. requirements specifications, architectural specifications, test plans and specifications, and many more.

One basic goal in software development projects is to keep artefacts consistent over project lifetime. Otherwise, the information content of these artifacts would decrease, which in turn could lead to delay or even errors during implementation due to contradictory and wrong understanding. Even testing does not help in such a situation, because running test cases that do not address the right requirements is waste. The result would be late delivery of the product or delivery of an erroneous product to the customer.

Changes to one particular artifact, e.g. a requirements specification, compromise consistency with other artefacts unless the project manager takes measures to restore consistency. The solely way to do so is to check for each change to a particular artifact which other artefacts also might be affected by this change. This is where the term "dependency" comes into play (as a means to support change management).

"*Dependency*" can be defined as "*relationship between two elements*". In the narrower sense, dependency means "*relationship between artefacts*" (like requirements, source code, and test cases). An artifact A is dependant on another artifact B if a change of B leads to a change of A.

Knowing about dependencies between artefacts in a software development project is important and necessary to preserve consistency of artefacts and thereby support efficient and effective development that means providing developers with up-to-date and consistent documentation (requirements, design, test cases) to avoid later rework or erroneous implementation, e.g implementation of a software that is not aligned to the latest customer requirements.

The following example aggravates the importance of dependencies in software development:

In safety-relevant projects like software for trains, busses, or spaceshuttles, it is extremely important that each requirement is tested appropriately. Otherwise, the life of passengers would be endangered. Usually, there is an external authority (like the German TÜV) that checks this and demands a proof that all requirements are covered by test cases. This is usually done by explicitly relating test cases to requirements so that it is traceable for the assessor which requirement is tested by which test case. This documentation of dependencies is a very important means to check and ensure safety of the endproduct.

The following sections outline the most important types of dependencies that exist in software development.

1.2.1 Dependencies between Artefacts

During the development lifecycle in a project multiple artefacts evolve, e.g., requirements describing the desired functionalities, design specifications describing the components that the software consists of, source code, and test cases, and lots of other documents like minutes of meetings etc. These artefacts do not exist in isolation from each other but depend on each other: architects derive design components from the requirements, developers create source code based on the design, and testers create test cases to test requirements, as depicted in Figure 1.

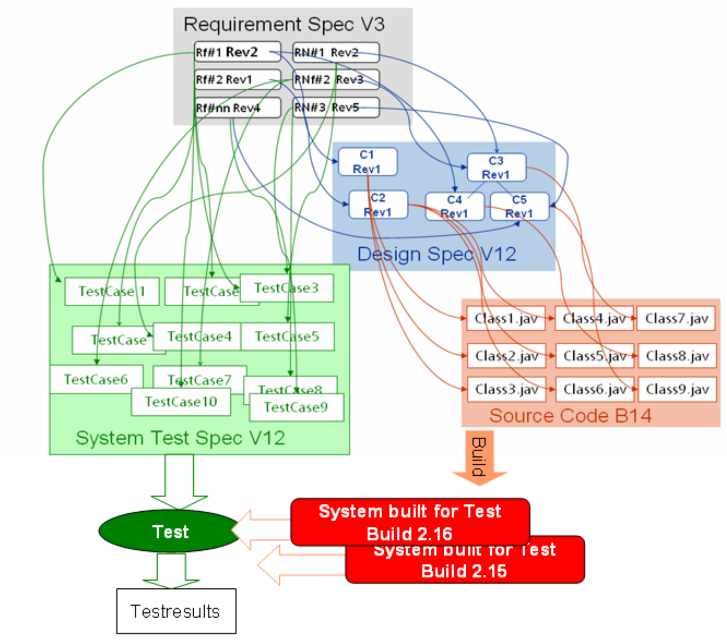


Figure 1 Dependencies between Artefacts

Knowledge about these dependencies helps project participants to keep the overview about the artefacts, e.g., keeping consistency of work products. Usually, project team members know about these dependencies because they “have these dependencies in their heads”. Making these dependencies explicit (e.g., by writing them into a table) helps to share knowledge about dependencies within the distributed team.

One important method to capture dependencies between artefacts explicitly is Requirements Tracing (RT). Traceability is „the degree to which a relationship can be established between two or more products of the development process“ (IEEE standard glossary). Numerous standards require traceability, e.g., ISO 15504, CMMI, IEEE Std830. Furthermore, many companies are mandated to implement traceability.

For GSD project managers (and other project team members such as architects, quality assurance managers, or developers/testers) management of these types of dependencies is important in order to be able to (a) conduct effective and efficient change impact analyses, (b) ensure consistent implementations of changed requirements, and (c) analyze quality assurance coverage of changed project artifacts, e.g., with test scenarios.

1.2.2 Dependencies between Team Members

Besides dependencies between artefacts there are also dependencies between people in a project, e.g., communication paths between project team members, as depicted in Figure 2.

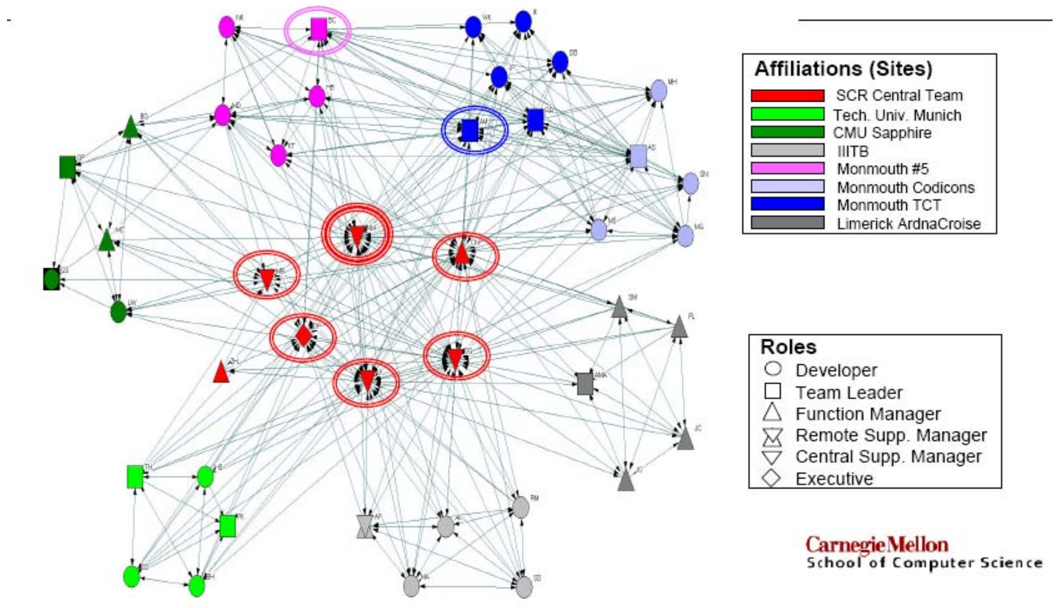


Figure 2 Dependencies between Project Team Members - Social Network Analysis [101]

Figure 2 shows a social network analysis [101] which aims at identifiying communication paths between project team members. In the figure there are 8 teams, each team consisting of a handful of people with particular roles. The lines between the team members indicate the communication paths. Such an analysis can be used to identify communication needs in distributed projects.

1.2.3 Dependencies between People and Artefacts

There are not only dependencies among artefacts, or dependencies among team members, but also between team members and artefacts. Such dependencies can be used to provide context information, as depicted in Figure 3. The figure displays for a particular task which context information for this task exists in terms of related artefacts and people that have been involved in the task. Such context information can be used by the team member who is responsible for this task, e.g., to answer questions, such as:

1. On which requirements is my current task based on? (This is called requirements awareness [26])
2. What is the value behind these related requirements?
3. Which stakeholder requested the requirements (source of requirements; may be helpful to know for inquiries)?
4. Which artefacts do I have to take into consideration because they are related to my task?
5. Who worked on preceding tasks? Who can I contact when problems occur with my task?

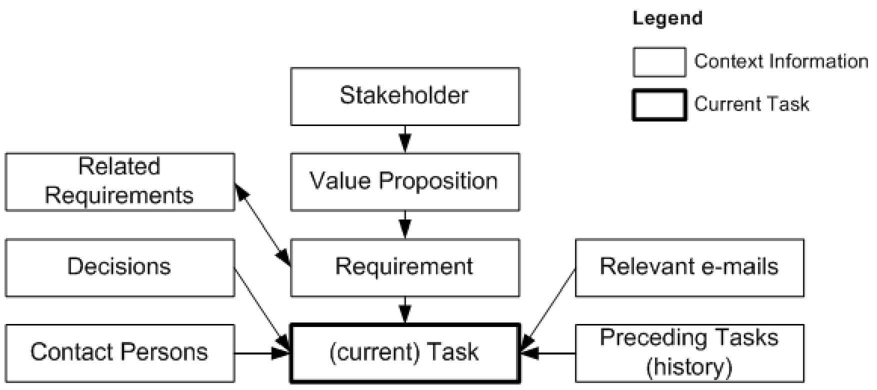


Figure 3 Dependencies between People and Artefacts

Context information is extremely important and harder to get in global software development projects due to the distribution of team members. Identification and management of the dependencies that exist between the elements in Figure 3 supports collaboration in global software development, as described below.

1.3 State of the Practice of Dependency Management

Existing approaches for dependency management work well in projects with a small number of requirements. These approaches do not scale in medium- and large-sized projects with a large number of requirements (hundreds, or even thousands of requirements), e.g. due to the explosion of efforts that would be necessary for dependency management.

The following subsections describe currently available dependency management approaches and explain why they are not sufficient.

1.3.1 Implicit Dependency Management

Developers develop source code based on given requirement descriptions. Thus, they usually know imlicitly, which requirements are implemented by which pieces in source code. Whenever a requirement changes, developers can apply this knowledge and adapt the relevant pieces of source code. Managing dependencies between requirements and source code in this way can be called "implicit dependency management".

The problem with implicit dependency management is that this knowledge about dependencies is not transparent and reusable by other project team members, e.g. when some developers collaborate on some source code. Furthermore, implicit dependency management does not meet the demands of software engineering standards like CMMI [102].

1.3.2 Explicit Dependency Management

The insufficiency of implicit dependency management resulted in the second approach for dependency management. The idea is to make knowledge about dependencies explicit by writing them down as cross-references [44] in documents, in tables, matrices [77] or requirements management tools. Thereby, dependency information gets usable for all project team members for applications like change impacts analysis, coverage analysis, or derivation analysis (as described in section 2.1). Furthermore, explicit dependency management is demanded by multiple software engineering standards like CMMI [102] and is extremely important especially in safety-critical projects where explicitly captured dependencies are a prerequisite for getting an approval from an external assessor, e.g. the German TÜV.

Explicit dependency management is a challenge because it needs significant effort to provide full traceability of all requirements to all other artefacts [51]. Furthermore, requirements tend to change and traceability information (traces) has to be maintained over time to keep its value, which might again cause extra effort and delay. This is especially true for manual dependency management approaches. Thus, there emerged several approaches to (semi-)automate dependency management during the last decade, e.g. by capturing dependencies between artefacts in an automated way. Examples for such approaches are [35][4][23][70] and are described in detail later (in section 3.1.6).

The problem with these automated dependency management approaches is that the quality of generated dependencies is questionable: evaluations of these approaches (e.g. reported in [4] and [23]) demonstrate that the completeness and correctness of generated dependencies is not yet sufficient, because either not all existing dependencies are captured or some dependencies are captured where there is no dependency. Finally, this reduces the project team members' trust in automatically captured dependencies and the application in practice.

Another problem of such dependency management tools is that they are separate tools and not directly integrated into the project's tool infrastructure. These often leads to a "yet another tool"-syndrome [97] and makes users refuse the tool. The lack of tool integrations is one more reason why desirable dependency management across tool borders is still not state of the practice [119] and [120].

1.4 Criteria for Good Dependency Management

Good dependency management aims at the following goals:

G1.*High trust of project participants in captured dependencies*: that means high-quality of captured dependencies, i.e. a high degree of completeness and very few falsely captured dependencies.

G2.*Effort saving for dependency management:* that means cheap capture and maintenance of dependencies (cheaper than with currently available approaches).

G3.*No extra tool for dependency management (e.g. particular dependeny capture tools):* Instead, support for dependency management should be integrated into the given project tool infrastructure.

The three goals are prerequisites to make dependency management feasible in practice. Thus, the aspired solution is a combination of manual and tool-supported dependency management: manually captured dependencies are usually more trustworthy than automatically generated ones, and an appropriate tool support helps to reduce dependency management efforts.

The success criteria for such an approach are:

(a) *Availability of dependency management planning support*: At project start requirements engineers usability define a tracing policy [122]. This tracing policy defines which dependencies should be captured when and how (by which tools) during the project. In order to support this planning step of dependency management, there should be a framework that allows estimating and comparing different dependency management approaches in advance. The optimal result of this planning step is a feasible and usable tracing policy that can be "really lived" in the project.

(b) *Management of dependencies according to their value*: Skipping some dependencies (not capturing them) is not allowed by standards like CMMI [102], but adjustments can be made to the level of precision with which these dependencies can be captured, e.g. dependencies between requirements and source code elements can be captured at class or method level. The level of precision has a direct impact on the effort necessary to capture dependencies. Thus, low-value dependencies should be captured on a low level of precision (with reduced efforts); high-value dependencies should be captured with a higher precision. This is aligned to Value-based Software

Engineering [12][10] principles, which are explained in more detail in section 2.3.

(c) *Cheap (feasible) capture of dependencies*: the effort for capturing dependencies should be as low as possible (without losing quality of captured dependencies). Value-based principles as well as appropriate tool support can help to achieve this goal, as explained in the next section.

(d) *Capture of dependencies across tool borders*: different roles use different tools in global software development projects. Each tool contains different types of data that have dependencies, e.g. requirements management tools contain requirement descriptions, test management tools contain test cases, development environment and configuration management tools contain pieces of source code. Integration of these tools is a prerequisite to allow dependency management for dependencies between requirements, source code, and test cases.

(e) *Usage of dependencies for multiple purposes*: The more applications exist for using captured dependencies, the more justified is the spent effort of capturing dependencies, e.g. using dependencies for both change impact analysis AND collaboration support (as outlined in the following section) increases the benefits of dependency management.

An approach that considers these criteria is an approach that supports cheap, valuable, and high-quality dependency management. The following section outlines a solution framework that explains how my research contributions contribute to the improvement of dependency management in global software development projects.

1.5 A Solution Framwork to outline my Research Approaches

Figure 4 gives an overview on entities of a global software development project and relevant parameters.

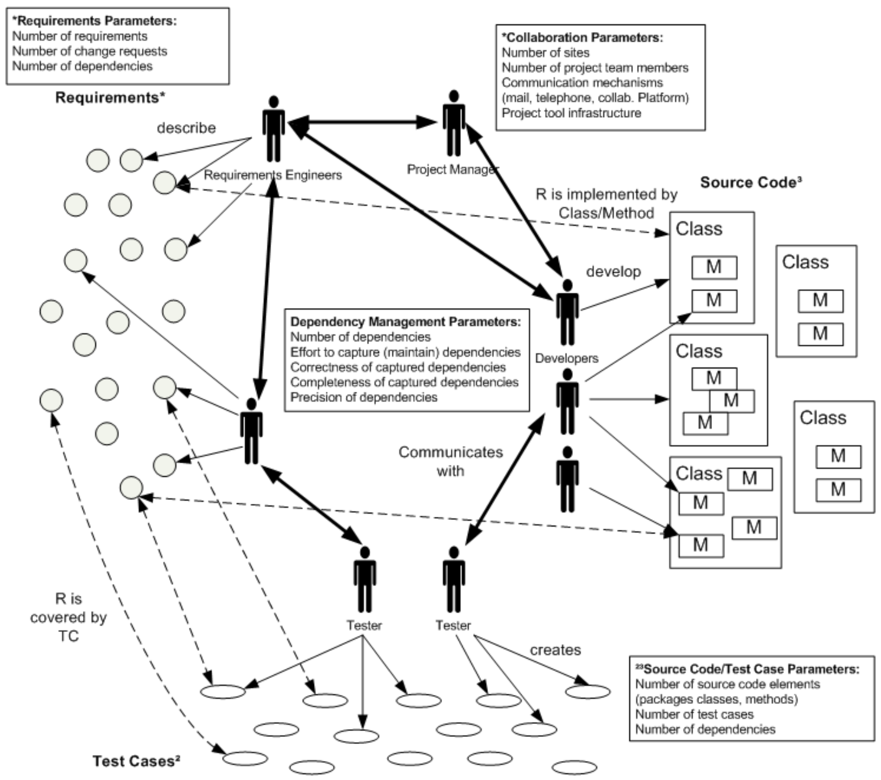

*Requirements Parameters:
Number of requirements
Number of change requests
Number of dependencies
Requirements*
*Collaboration Parameters:
Number of sites
Number of project team members
Communication mechanisms
(mail, telephone, collab. Platform)
Project tool infrastructure
describe
Requirements Engineers
Project Manager
Source Code³
R is implemented by Class/Method
Class
M
develop
Developers
Dependency Management Parameters:
Number of dependencies
Effort to capture (maintain) dependencies
Correctness of captured dependencies
Completeness of captured dependencies
Precision of dependencies
Communicates with
R is covered by TC
Tester
Tester
creates
²³Source Code/Test Case Parameters:
Number of source code elements
(packages classes, methods)
Number of test cases
Number of dependencies
Test Cases²

Figure 4 Dependency Manangement Entities

Entities comprise artefacts like requirements, source code, test cases; and people like project manager, requirements engineer, developers, and testers.

Dependencies exist between these entities, e.g. between requirements and test cases, requirements and source code classes or methods (slashed lines in Figure 4); Furthermore, the information who worked on which artifact can also be interpreted as dependency between a project team member and a particular artifact (thin arrows in Figure 4); finally there are also dependencies between project team members. They represent communication paths (thick arrows in Figure 4).

Regarding dependency management there are some parameters that have an influence on the applicability and cost-benefit of dependency management approaches like number of sites and project participants, number of requirements, communication mechanisms available in the project, (e.g., telephone, mail, collaboration platforms), project tool infrastructure (e.g. tools used for requirements management, development, test management, project planning).
Figure 5 contains further factors of dependency management. These are the major factors I focused on within my research.

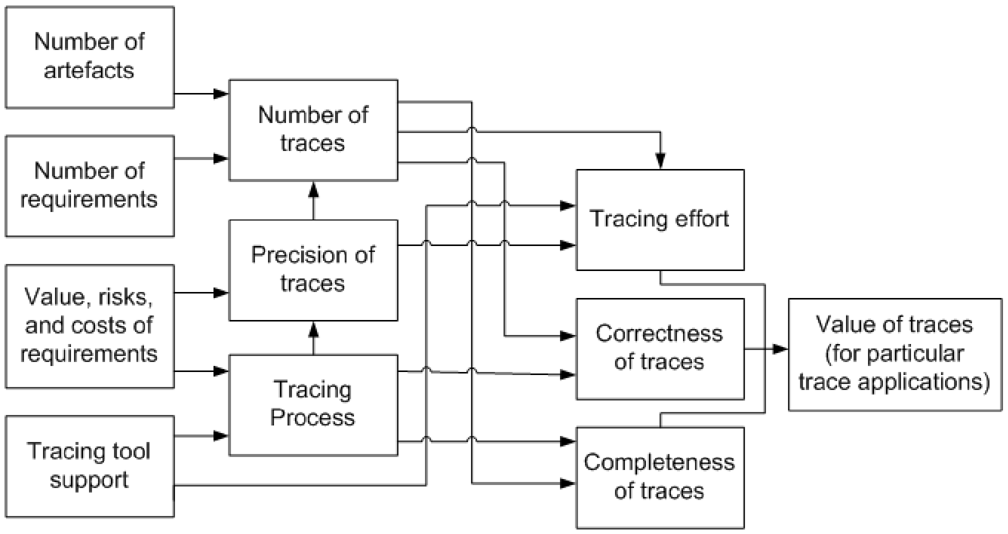


Figure 5 Major factors of dependency management

The left side of Figure 5 shows project-related parameters: *number of artefacts* (e.g. source code elements, test cases), *number of requirements*. Requirements usually have a priority. This priority can be defined by *value, risk (volatility), and costs of requirements*. Another project parameter is the *tool support* available in the project for tracing activities. Tool support ranges from requirements management tool that support manual capturing of dependencies to tracing tools that automate capturing of dependencies. These project-related parameters directly influence dependency management parameters:

(a) *Number of traces*: the higher the number of requirements and artefacts the higher is the number of potential traces (captured dependencies) between them.

(b) *Precision of traces*: is the level of detail of captured traces, e.g. requirements can be traced into source code at package, class, or method level. Precision depends on the chosen tracing process. For example, a value-based requirements tracing process (explained in detail below), takes the value, risk and costs of requirements as input to tailor the precision of traces for each requirement: high-priority requirements will be traced with a high level of precision (on method level), medium- and low-priority requirements will be traced with lower levels of precision (on class and package level). Precision influences the number of traces: one trace from a requirement to a source code class may substitute multiple traces from this requirement to source code methods within this class.

Furthermore, a lower precision reduces tracing efforts, because it is cheaper to trace requirements to class level than to method level.

(c) *Tracing process*: the tracing process depends on cost-benefit considerations. It may be influenced by parameters like requirement priorities and the tool support given in the project. Both parameters influence the feasibility of dependency management.

The core parameters of dependency management that define the overall cost-benefit of dependency management approaches are:

(d) *Tracing effort*: is the effort needed to capture and maintain dependencies between requirements and other artefacts.

(e) *Correctness of traces*: a correct trace is a captured dependency that stands for a true dependency between two elements. The opposite is a wrong trace, which is a captured dependency between two elements that do not have a dependency.

(f) *Completeness of traces*: means that all dependencies that exist between given elements are captured. Usually, a particular percentage of completeness can be achieved by manual or automated dependency management approaches.

Tracing effort, correctness and completeness of traces determine the cost-benefit of dependency management and thereby the *value of traces* for certain trace applications like change impact analysis.

By using the framework of project entities and parameters described above, my overall research goal is to meet the criteria for good dependency management approaches and to improve the cost-benefit of dependency management approaches.

The scope of my research is medium- and large-size GSD projects. These types of projects make it necessary to improve dependency management approaches due to increased complexities (higher number of requirements, project participants, and different tools used). For other project types, current dependency management approaches seem to be sufficient.

I try to reach my research goal by providing research contributions in the following areas:
1) *Planning dependency management*: I provide a tracing activity framework that supports a project manager in (a) defining which dependencies should be managed and (b) comparing dependency management approaches in advance (before dependency management starts in the project) to find the most cost-effective approach.
2) *Capturing dependencies explicitly, systematically, and consistently*: I provide a value-based requirements tracing approach to focus dependency management efforts on high-value dependencies. Furthermore, I provide a prototype implementation of a tool integration that supports capturing dependencies across tool borders (between a requirements management tool and a development environment).
3) *Application of explicit dependencies (traces)*: I provide a concept for a dependency-based notification system that supports communication and in-time notification of GSD project team members.

The following subsections outline our research areas and outline our research questions and approaches (within the areas of dependency planning, explicit capture of dependencies, and application of traces) and outlines how they influence/improve particularly the dependency management capability, but also the other particular capabilities necessary for good performance of GSD projects.

1.5.1 Research Area: Planning of dependency management

There are lots of dependencies in a typical GSD project (see above for types of dependencies). Furthermore, [112] defined traceability models that contain types of dependencies (captured and maintained as traces) that provide certain benefits to the project participants. Figure 6 depicts a simple low-end traceability model with a limited set of traces. In contrast to that, Figure 7 depicts a high-end traceability model that contains a bigger set of traces.

Low-end traceability models, as in the example above, establish relationships from requirements to the system components and other artefacts that satisfy those requirements. By capturing which components satisfy various requirements and which requirements are mapped to different components, the project team is able to verify that all requirements are addressed by the system.

In the test phase of a software project low-end users use requirements traceability to maintain links between test cases and requirements in order to understand which test cases address every requirement.

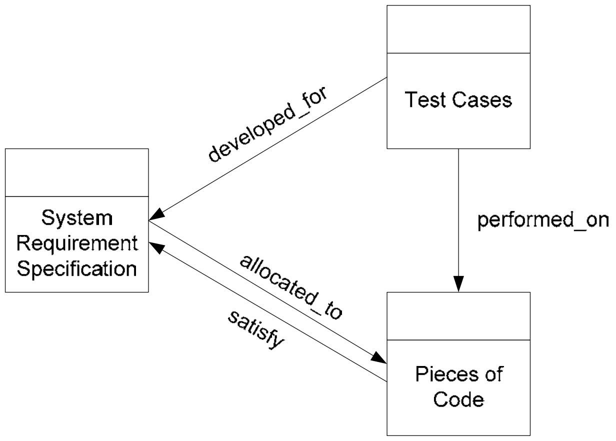


Figure 6 A low-end traceability model [112].

High-end users of traceability employ much richer traceability schemes than low-end users and also use traceability information in much richer ways [112]. Rationale Management is a central task in high-end traceability models. As depicted in Figure 7, high-end traceability models do not only store information about test cases and code, but also information about resources, rationale, decisions, and alternatives.

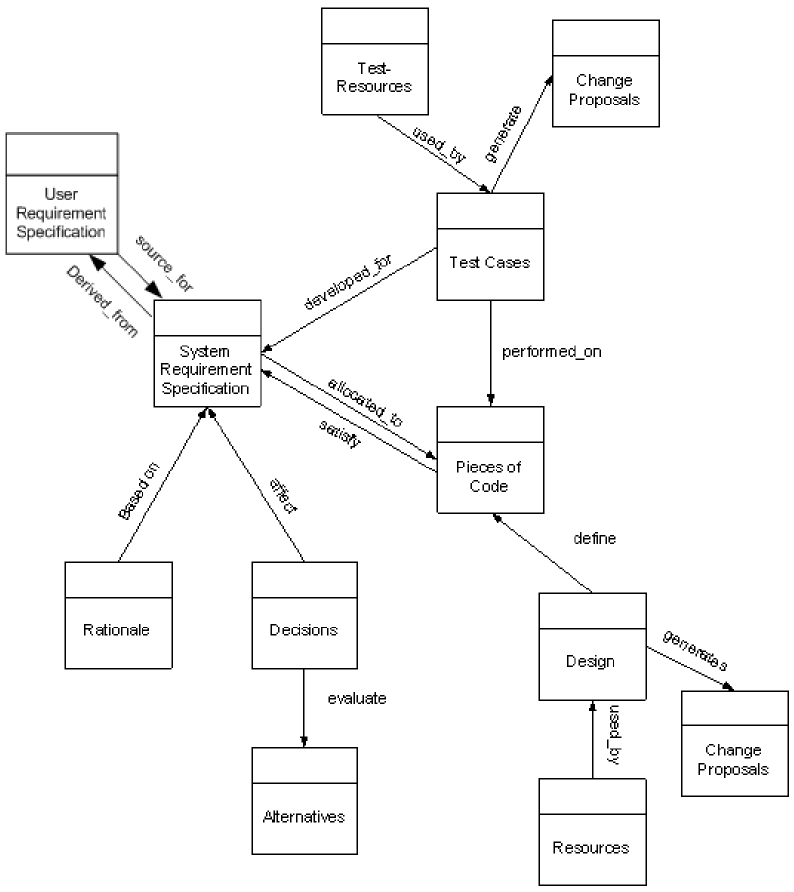


Figure 7 A high-end traceability model

Managing all these dependencies would cause considerable high efforts and tool challenges (which dependencies should be maintained with which tools?).

Thus, it is necessary to define at the beginning of the project, which data is useful for dependency management (based on parameters like the used project process, roles, tasks and artefacts in the project) and which dependencies are valuable to be captured and maintained, e.g., project managers should focus on high benefit and high risk dependencies.

Managing the right dependencies would bring certain benefits to particular project participants, such as:

- Project Managers: who could use dependency information for reporting of project progress based on requirements/functions developed that have successfully passed test with sufficient coverage; or for reporting on change impact analysis (effort, delay, risk).
- Architects: who need dependency information to perform technical change impact analysis (to identify components that need to be checked, changed, newly written, exchanged).
- Quality Assurance (QA): who use dependency information to analyze test coverage of interfaces, control structures, model nodes and links (state charts, sequencen charts, source code blocks)

As basis for planning dependency management activities in a GSD project, I developed a Tracing Activity Framework (TAF) that defines a unified terminology for tracing activities and parameters, and maps existing terms to the framework. TAF consists of tracing activities like trace generation, deterioratiation, validation, rework, and trace usage. For each tracing activity, the relevant parameters that influence the activity are assigned. Thus, TAF is a good means to model requirements tracing alternatives by using the defined activities and parameters. I evaluated TAF with an initial case study in a large Austrian bank where I modeled 3 different tracing alternatives and used TAF to compare the efforts and the overall cost-benefit of each alternative in order to find the most suitable one. Thus, it is also a good means to plan dependeny management at the beginning at the project: it can be used to calculate the number of traces to be established and maintained and supports the reasoning on which traces have priority and can be done with certain budget. TAF is explained in detail in section 4.1.

TAF is a categorization of tracing-related activities and parameters and thus is very well useable as a framework for a systematic literature review. As fundament of this work I performed a systematic literature review about currently existing dependency management and requirements tracing approaches. Besides the identified approaches, one main observation was that there is no process model for tracing, e.g., containing a planning step to decide on the dependencies to be managed. The literature survey and its results are described in section 3.1.

1.5.2 Research Area: Capturing dependencies explicitly, systematically and consistently

Knowledge about technical dependencies is usually stored in the heads of project members, which may seem cheap, but this knowledge is not effectively or efficiently available for other project participants. Therefore, requirements tracing approaches have emerged with the goal to explicitly capture and model dependencies between requirements and artefacts like design, source code, and test cases as traces, mostly with little or no tool support.

The main drawback of a manual tracing approach is high effort (imagine tracing thousands of requirements to hundreds of source code classes), which in practice leads to (a) rather sporadic and unsystematic trace capture (which in turn often leads to insufficient quality of traces for dependency analysis) and (b) high effort to motivate personell to capture traces (which in turn leads to delay and high costs, and again bad trace quality, if motivation is not too high).

Thus, research approaches emerged to automate trace generation in order to reduce tracing effort. Up to now, these approaches continue to suffer from (a) low trace quality, and (b) insufficient integration of tool support for tracing so available information on dependencies in data repositories of different tools is hard to access and integrate for dependency analysis.

I tackle these issues for managing technical dependencies with the following approaches:

(a) *Value-based requirements tracing* systematically supports project managers in tailoring requirements tracing precision and effort based on the parameters stakeholder value, requirements risk/volatility, and tracing costs to achieve high-benefit traceability information with limited budget. Value-Based Software Engineering as discipline is described in section 2.3. Value-Based Requirements Tracing and some aspects (correctness of traces, precision of traces) that I analysed in detail are described in sections 5.1, 5.2, 5.3, and 5.4.

(b) *Integrated Developer Tool support for dependency management* reduces the effort for capturing dependencies between requirements and source code elements significantly by providing semi-automated support for users and thereby improves trace quality such as correctness and completeness of captured traces. Furthermore, traces have to be (re-)checked after a change and this approach also supports trace maintenance during system evolution. Integrated developer tool support is described in section 5.5.

1.5.3 Research Area: Application of traces for dependency management in project and product management scenarios

Particular roles in a GSD project need particular information to perform a particular task. One part of this information is context information, as depicted in Figure 2, Figure 3, or Figure 6: such context information consists of requirements that stand behind the given task, tasks that have been performed by other project participants predecessing to the current one, task-related communication that has been performed and decisions that have been made before. Documents that contain such context information, like emails, minutes of meetings, specification documents, or people that are or have been involved into the current task (or predecessing ones) can be related to the current task by setting-up dependencies in between (also called social or socio-technical traces) and providing a tool support that allows the retrieval of the context information later by using these dependencies.

Capturing these traces explicitly would improve the quality of task fulfillment, because important information, such as relevant decisions and rationale could be easily provided to the one who is performing a task.

Currently in GSD projects, such context information is usually available in separated tools, e.g., a project manager uses a requirements management tool to specify a concrete requirement and can maintain additional requirements-related information in this tool. Or a tester manages the data and information necessary for his testing tasks in a test management tool.

The problem arises, when a tester needs additional information that cannot be retrieved from the test management tool, because it is stored in a different tool at another site or a project participant at another site could help. In such situations, the current approach to get this context information is to write an email (which can cause delay) or call the colleague at the other site (this option is limited by time-zone differences and the availability of colleagues at other sites)

Options that are often used to address these problems are:

- definition of work packages appropriate for distributed working (low coupling), e.g. putting packages with high communication effort together at one site, proper definition of interfaces and roles, and regular face-to-face meetings for discussions of interfaces between work packages (in case of major changes)
- Groupware tool support for organizational dependencies such as Netmeeting, Wikis, collaboration platforms with e-mail notification support so that relevant information of one site is automatically transported to other sites on certain events, e.g., document check-ins, etc.

Weaknesses of these approaches are that face-to-face meetings are expensive to establish and that current notification support generates many email events for changes (e.g. in Trac) and nobody is reading that due to low content information (consequence: people may take lot of effort for trivial things or choose to ignore or need to filter). Furthermore, there are weaknesses regarding user interface and customization capabilities of such tools.

I tackle the issues of currently available approaches mentioned above for managing social dependencies by providing the concept of a *role-based in-time notification system*. The idea of a role-based notification system is to complement the limited synchronous communication in GSD projects with asynchronous communication via the tool set used in a project that means that most of the information necessary for a project team member should be provided via the tools that he uses. Tool integration is again a prerequisite here in order to be able to retrieve relevant information from other tools in the project's tool set. Using traces in this context is a means to define which information is relevant and related to the current task of a project team member. By using this network of traces the project team members should be notified about events (changes) happening somewhere in the project that affect his current task.The number of notifications scales down to a degree where only really relevant notifications for a role in a particular situation are delivered. This role-based in-time notification system is based on the Application Lifecycle Management (tool integration) concept and supports across-the-tools notifications. The notification system approach is explained in detail in section 6.

As evaluation concept for the contributions in this work I use empirical case studies and report their results.

2 Fundaments of this Work

This chapter describes global software development, requirements engineering, value-based software engineering and application lifeycle management as fundaments of this work.

2.1 Global Software Development

Global Software Development (GSD) can be defined as "software work undertaken at geographically separated locations across national boundaries in a coordinated fashion involving real time (synchronous) and asynchronous interaction" [94].

It involves (a) communication for information exchange, (b) coordination of groups, activities and artifacts so that they can contribute to the overall objective, and (c) control of groups (adhering to goals and policies) and artifacts (quality, visibility & management).

The main reasons for performing GSD projects are [94]:

- New markets or presence in the new markets
- Competitive advantage: cheap resources
- "Follow-the-sun-development", where work tasks are shifted around the globe from one site to another so that it can be worked on round-the-clock.
- Acquisitions & mergers with new sites as a result.

Multiple issues of GSD are reported in literature. The following list provides a collection and overview of issues [94]:

- Strategic issues: when to allocate a task to whom and how;
- Communication and coordination issues: due to distance, time zone difference, infrastructure support, distinct backgrounds, lack of informal communication;
- Cultural issues: power distance, individualism vs. collectivism, attitude to time etc.;
- Geographical dispersion: vendor support, access to experts, software practices that need face-to-face communication;
- Technical issues: information and artifact sharing, software architecture;
- Knowledge management: slow communication, poor documentation, tacit knowledge, repositories etc.;

Further descriptions of GSD and its issues can be found in [67] and [26].

Several stakeholders are typically involved in GSD projects, each having certain stakeholder interests. Typically, there is a:

- *Business unit manager*: main interests are economic concerns, such as cost-competitive use of scarce resources in a global resource pool and establishing market proximity, e.g., by having a project site in China;
- *Project manager*: the main interests of a project manager in a GSD project is to ensure efficient and effective collaboration of all project participants (i.e. his project team consisting of architects, developers, and testers). Furthermore, project planning and tracking is an important issue, as well as change management, which deals with planning, controlling and consistently implementing changes to the software to be delivered;
- *Quality assurance manager*: is responsible for the quality of the processes used in the project, e.g., change management processes, and the quality of the delivered product.

Thus, the following capabilities are demanded of GSD project participants in order to successfully reach the project goals and the stakeholder interests:

First of all, the value propositions of the success-critical stakeholders, e.g. the customers who pay for the software or the users who will have to work with it, have to be elicited and managed. This is a prerequisite for project success (that every success-critical stakeholder gets the piece of software that he really needs). *Value-Based Software Engineering* (see section 2.3 for details) is a capability that supports software engineers (analysts) to elicit, analyze, negotiate, and monitor the stakeholders' value propositions.

Risk Management (as capability) is indispensable in software engineering projects to proactively identify risks that might cause project failure, analyze them and develop countermeasures to mitigate or reduce them.

Collaboration capabilities are a crucial point in GSD. They depend very much on the project participants' skills, the collaboration technology (e.g., collaboration platforms, or a common document repository), and the available ways of communication (e.g., telephone, mail, netmeeting, face-to-face meetings). *Having a common process* is important, because distributed teams need a common process for collaboration even more than collocated project teams.

Keeping the overview on the developed product, the project activities, and project progress is a fundamental capability necessary in GSD projects (especially for the project manager). Due to the distributed sites it is often hard for project managers to gain visibility into the project status and thus there are often justified concerns that things would not be delivered in a timely way (as reported by [67]). In this context, systematic change impact analysis is also important. Software development projects are lively and changes to some artefacts are inevitable. The outcome of this is that changes of one artifact may lead to necessary changes of other artifacts. *Change impact analysis* ensures that changes are implemented consequently and consistently in all affected artefacts.

Application Lifecycle Management is a capability that is necessary in GSD to provide a homogeneous, integrated tool infrastructure, so that each role in the project can manage relevant data and dependencies in between.

One elementary capability that supports multiple of the ones described above is *dependency management*: GSD projects are complex projects and the vast amount of dependencies that typically exist in such projects are one major reason for a project's complexity. Dependencies exist between artefacts, e.g, between requirements and other artefacts) or between work tasks that have to be done at multiple different sites in the project, e.g, development and test tasks of a version "XY" of a software module.

Management of dependencies that exist (a) on a technical level, e.g., between requirements, source code elements, or test cases, or (b) on a social level, e.g., dependencies between colleagues that work together on the same component supports the project team in their collaboration and in other tasks such as change impact analysis or coverage analysis and allows the project manager to follow the life of a requirement throughout the software development lifecycle.

Identifying these dependencies and capturing them explicitly is a prerequisite for efficient and effective collaboration of the GSD project team. Furthermore, it supports requirements and change management and keeping the overview in the project.

As indicated above, dependency management is an area that provides multiple benefits in GSD projects, such as ensuring high process and product quality (e.g., consistency of artefacts) or increased cost efficiency of important tasks like change impact analysis or coverage analysis. Thus, depenency management is worthwile to be investigated in detail. The following section motivates our research and outlines our research contributions.

Multiple types of dependencies exist in GSD projects. The following subsections describe tasks in GSD projects where the usage of dependencies makes the tasks more efficient and effective.

Change Impact Analysis - "What if this were to change?"

A change management process is a crucial sub-process of software engineering. It describes the procedure how to propose, analyse, decide upon, and implement a particular change request.

The most important part of such a change management process is the analysis step where project managers and architects analyze the feasibility, effort and the consequences of the proposed change request. Change impact analysis is one key task during analysis: It identifies which consequences a changing artifact has on other artefacts, i.e. which impact a changing artifact has on other artefacts. Capturing interdependencies between artefacts explicitly as traces supports change impact analysis, because the change analyst just has to follow the traces from the changing artifact to other artefacts that probably have to be adapted. The challenge herewith is to have correct and complete sets of traces in order to support a complete change impact analysis. The basic hypothesis is that (despite the extra effort for capturing traces) trace-supported change impact analysis is in total cheaper than change impact analysis without trace support.

Coverage Analysis - "Have I covered everything?"

Coverage analysis is a means to ensure consistency of different models that exist in a project. Imagine there are a requirements model, an implementation model, and a test model. Coverage analysis means, for example, to check if there is at least one test case in the test model for each requirement in the requirements model. Each requirement has to be covered by at least one test case. This is of great importance, especially in safety-critical projects where the project manager has to prove that all requirements are covered by test cases and where an external authority, such as an assessor from an technical inspection agency, checks this coverage.

Responsibility Documentation (Derivation Analysis) - "Why is this here?"

Some artefacts can be derived from other artefacts, e.g., concrete functional software requirements can be derived from high-level user requirements, which in turn have been derived from the customer's business needs. Furthermore, there is a particular stakeholder behind each requirement. Responsibility documentation means to capture which requirements are derived from which other requirements and who is the source of each requirement. For example, having minutes of meetings where it is written down which stakeholder requested which requirement helps later to discuss with the customer when he (for whatever reasons) does not remember to have requested the requirement.

Effective communication in GSD projects

When project team members work on a particular task there are usually multiple relations to other team members and artefacts that have to be taken into consideration, as depicted in Figure 3. Knowing about these dependencies, e.g., who worked on this task before or who else is affected in his work when I have finished my task, helps project team members to have the complete work context to finish a task successfully. Dependencies (captured explicitly as traces) between all kinds of artefacts and people can be used to support timely and effective information exchange between the related team members, as it will be described later in detail.

Requirements engineering is a discipline that is regarded as extremely crucial for project success by both practitioners and researchers. It is outlined in the following subsection. The focus on this description is on requirements management and requirements tracing as means for managing dependencies.

2.2 Requirements Engineering

Requirements engineering is a discipline that deals with understanding, documenting, communicating and implementing customers' needs. The Standish Group reports in a study from 1995 (study data still seems to be true today) that 30 % of Software development projects fail, and that 70 % of the remainder is over budget or behind schedule. More than 50 % of this trouble is caused by inadequate requirements definition, such as:

- Lack of user input
- Incomplete requirements/specifications
- Changing requirements/specifications

Thus, insufficient understanding and management of requirements is seen as the biggest cause of project failure.

In order to improve this situation a systematic process to handle requirements is needed. A typical Requirements Engineering process is depicted in Figure 8. The main steps are:

- Requirements Elicitation: collecting requirements from sources like stakeholders, existing documentation of similar products, etc.
- Analysis of requirements: negotiation of requirements between the stakeholders to come to a common understanding and prioritization of requirements to decide which requirements have to be implemented for the first release of the product and which requirements in subsequent releases;
- Requirements specification: create requirements documents that describe all requirements;
- Requirements validation: checking if the requirements are aligned to the real customer needs.

These 4 steps can be summarized as requirements development, which is out of scope of this work. More about requirements development, e.g., concrete methods, can be found in [122] or [52].

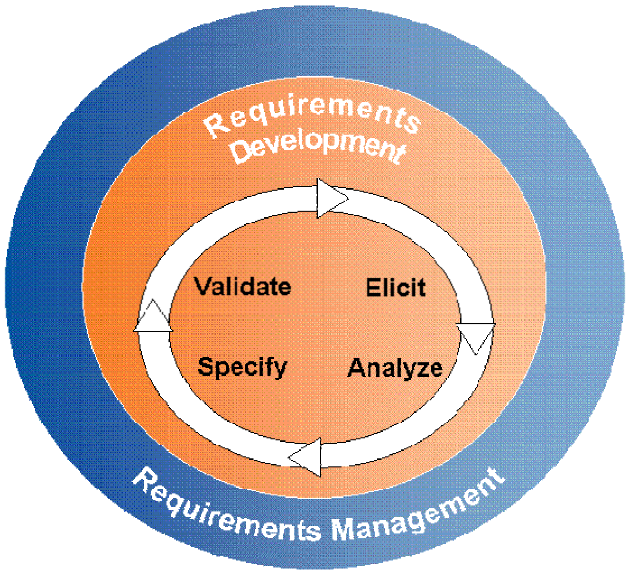


Figure 8 A Requirements Engineering Process [52]

Requirements Management is a "supporting discipline" to control all requirements and their changes during the development life cycle and to identify and resolve inconsistencies between the requirements and the project plan and work products. One important method of requirements management is requirements tracing.

The [43] provides the following definition of traceability:

"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match"

An analysis of the traceability problem is published by Gotel [51]. Watkins [133] describes why tracing can help the project manager in verification, cost reduction, accountability, change management/maintenance, identification of conflicting requirements and consistency checking of models.

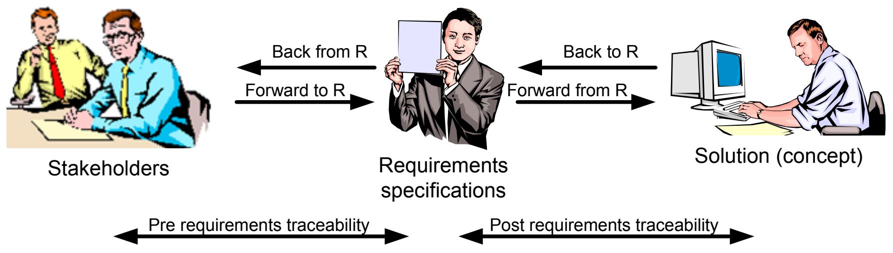

Back from R
Forward to R
Back to R
Forward from R
Stakeholders
Requirements specifications
Solution (concept)
Pre requirements traceability
Post requirements traceability

Figure 9: Requirements traceability links

There are four typical kinds of traceability links (see Figure 9):

- **Forward from requirements**: Responsibility for requirements achievement must be assigned to system components, so that accountability is established and the impact of requirements change can be evaluated.
- **Backward to requirements**: Compliance of the system with requirements must be verified, and gold-plating (designs for which no requirements exist) must be avoided.
- **Forward to requirements**: Changes in stakeholder needs, as well as in technical assumptions, may require a radical reassessment of requirements' relevance.
- **Backward from requirements**: The contribution structures underlying requirements are crucial in validating requirements, especially in highly political settings.

If requirements change, the impacts of these changes can be analyzed by using traces.

Figure 10 illustrates useful traces:

- Traces between requirements and design elements: in order to identify which requirement will be contained in which design component.
- Traces between design elements and code pieces: requirements can be traced into source code by using traces between requirements and design components, and design components and source code.
- Traces between requirements and test cases: test cases are used to find out if a requirement is well defined or if some information is missing. If you cannot create a test case for a requirement, you haven not all information you need for completely defining the requirement.

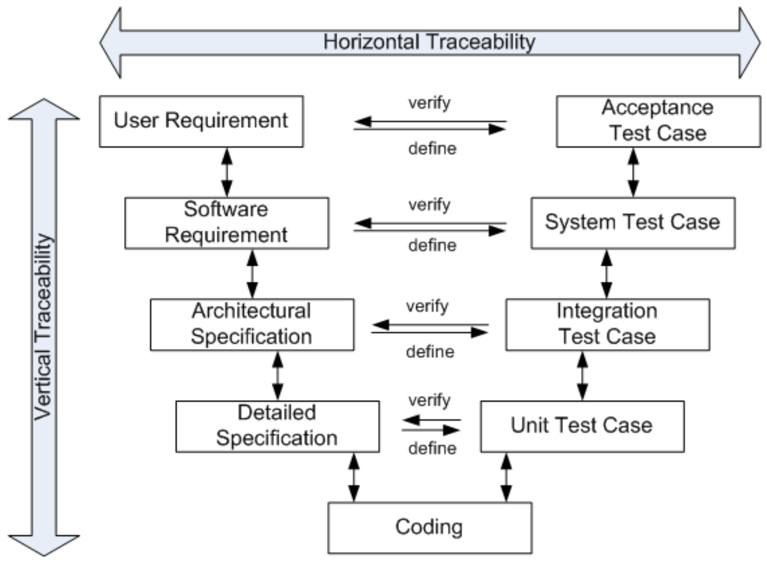


Figure 10: Traces between artefacts [121]

Furthermore, traces should be bidirectional. This means that you can use traces both in a backward and forward direction, e.g. forwards from artefact A to artefact B and backwards from artefact B to artefact A. Traces can help to find out which artefacts will need reworking if another artefact changes. These traces from an affected artefact to another one are then called "suspect traces".

Traces can be generated manually or automatically e.g. by parsing manually inserted requirements keys in source code files. Furthermore, they can be captured at different levels of granularity resulting in different efforts for trace generation but also in different value contributions, e.g. tracing into source code at method level is generally more expensive than tracing at class level. On the other hand, it provides more detailed information of where a requirement is implemented.

2.3 Value-Based Software Engineering

Following [12] and [10], the motivation for Value-Based Software Engineering can be described as follows: much of current software engineering practice and research is done in a value-neutral setting, in which:

- Every requirement, use case, object, and defect is treated as equally important;
- Methods are presented and practiced as largely logical activities involving mappings and transformations (e.g., object-oriented development);
- "Earned value"-systems track project cost and schedule, not stakeholder or business value;
- A "separation of concerns" is practiced, in which the responsibility of software engineers is confined to turning software requirements into verified code.

In earlier times, when software decisions had relatively minor influences on a system's cost, schedule, and value, the value-neutral approach was reasonably workable. But today and increasingly in the future, software has a major influence on most systems' cost, schedule, and value; and software decisions are inextricably intertwined with system-level decisions.

Also, value-neutral software engineering principles and practices are unable to deal with most of the sources of software project failure. Major studies such as the Standish Group's CHAOS report [128] find that most software project failures are caused by value-oriented shortfalls such as lack of user input, incomplete requirements, changing requirements, lack of resources, unrealistic expectations, unclear objectives, and unrealistic time frames.

Further, value-neutral methods are insufficient as a basis of an engineering discipline. The definition of "engineering" in Webster's dictionary is "the application of science and mathematics by which the properties of matter and sources of energy in nature are made useful to people." Most concerns expressed about the adequacy of software engineering focus on the shortfalls in its underlying science. But it is also hard for a value-neutral approach to provide guidance for making its products useful to people, as this involves dealing with different people's utility functions or value propositions.

This situation creates a challenge to the software engineering field to integrate value considerations into its principles and practices. Here are the seven key elements that provide candidate foundations for VBSE:

1. Benefits Realization Analysis
Many software projects fail by succumbing to the "Field of Dreams" syndrome. This refers to the American movie in which a Midwestern farmer has a dream that if he builds a baseball field on his farm, the legendary players of the past will appear and play on it ("Build the field and the players will come"). In The Information Paradox [129], John Thorp discusses the paradox that organizations' successes in profitability or market capitalization do not correlate with their level of investment in information technology (IT). He traces this paradox to an IT and software analogy of the "Field of Dreams" syndrome: "Build the software and the benefits will come".
To counter this syndrome, Thorp and his company, the DMR Consulting Group, have developed a Benefits Realization Approach (BRA) for determining and coordinating the other initiatives besides software and IT system development that are needed in order for the organization to realize the potential IT system benefits.

2. Stakeholder Value Proposition Elicitation and Reconciliation

It would be convenient if all the success-critical stakeholders had readily expressible and compatible value propositions that could easily be turned into a set of objectives for each initiative and for the overall program of initiatives. “Readily expressible” is often unachievable because the specifics of stakeholders’ value propositions tend to be emergent through experience rather than obtainable through surveys. In such cases, synthetic-experience techniques such as prototypes, scenarios, and stories can accelerate elicitation.

Readily compatible stakeholder value propositions can be achievable in situations of long-term stakeholder mutual understanding and trust. However, in new situations, just considering the most frequent value propositions or success models of the most frequent project stakeholders (users, acquirers, developers, maintainers) shows that these are frequently in conflict and must be reconciled.

For example, Figure 11 shows a “spider web” of the most frequent “model clashes” among these stakeholders’ success models.

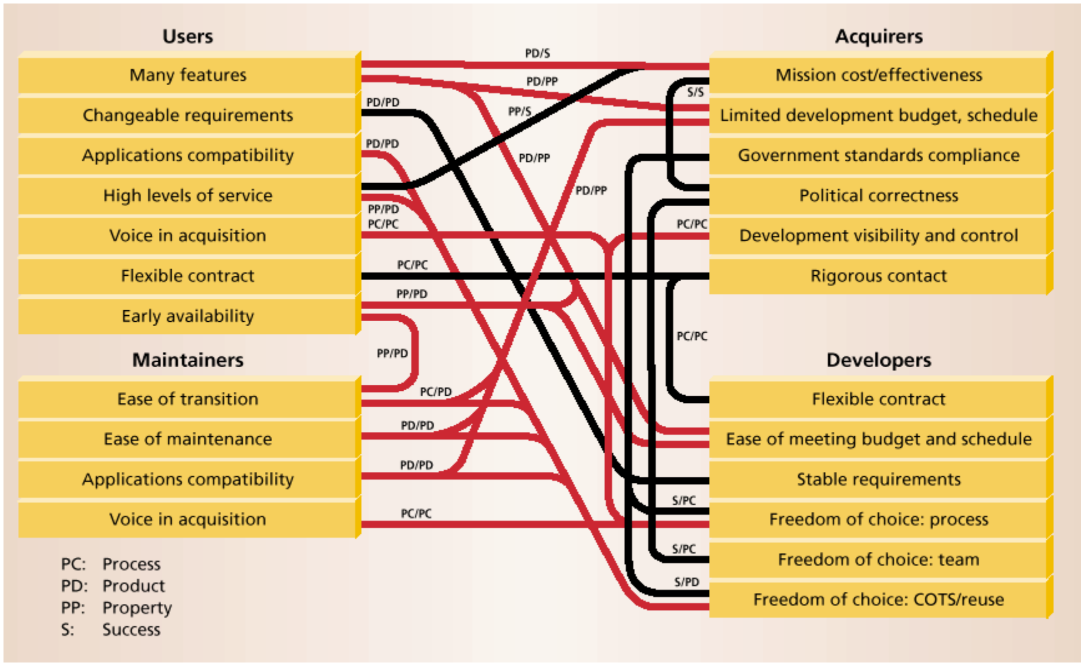


Figure 11 Model clash diagram [10]

The left- and right-hand sides of Figure 11 show these mostfrequent success models. For example, users want many features, freedom to redefine the feature set at any time, compatibility between the new system and their existing systems, and so on.

However, the Spiderweb diagram shows that these user success models can clash with other stakeholders’ success models. For example, the users’ “many features” success model clashes with the acquirers’ “limited development budget and schedule” success model, and with the developer’s success model, “ease of meeting budget and schedule.”

The developer has a success model, “freedom of choice: COTS/reuse” that can often resolve budget and schedule problems. But the developer’s choice of COTS or reused components may be incompatible with the users’ and maintainers’ other applications, causing two further model clashes. Further, the developer’s reused software may not be easy to maintain, causing an additional model clash with the maintainers.

The red lines show the results of one of the analyses performed in constructing and refining the major model clash relationships. It determined the major model clashes in the Bank of America Master Net development, one of several major project failures analyzed.

Given the goodly number of model clashes in Figure 11 (and there are potentially many more), the task of reconciling them may appear formidable. However, there are several effective approaches for stakeholder value proposition reconciliation, such as:

- Expectations management. Often, just becoming aware of the number of potential stakeholder value proposition conflicts that need to be resolved will cause stakeholders to relax their less-critical levels of desire. Other techniques such as well-calibrated cost models and "simplifier and complicator" lists help stakeholders better understand which of their desired capabilities are infeasible with respect to budget, schedule, and technology constraints.
- Visualization and tradeoff-analysis techniques. Frequently, prototypes, scenarios, and estimation models enable stakeholders to obtain a better mutual understanding of which aspects of an application are most important and achievable.
- Prioritization. Having stakeholders rank-order or categorize the relative priorities of their desired capabilities will help determine which combination of capabilities will best satisfy stakeholders' most critical needs within available resource constraints. Various techniques such as pairwise comparison and scale-of-10 ratings of relative importance and difficulty are helpful aids to prioritization.
- Groupware. Some of those prioritization aids are available in groupware tools, along with collaboration-oriented support for brainstorming, discussion, and win-win negotiation of conflict situations.
- Business case analysis. Determining which capabilities provide the best return-on-investment can help stakeholders prioritize and reconcile their value propositions. Business case analysis is discussed in more detail next.

3. Business Case Analysis

In its simplest form, business case analysis involves determining the relative financial costs, benefits, and return on investment (ROI) across a system's life-cycle as:

$$ROI = (Benefits - Costs)/Costs \qquad (1)$$

Since costs and benefits may occur at different times, the business case analysis will usually discount future cash flows based on likely rates of interest, so that all of cash flows are referenced to a single point in time (usually the present, as in Present Value).

One can then compare two decision options A and B in terms of their ROI profiles versus time. In Figure 12, for example, Option A's ROI becomes positive sooner than Option B's ROI, but its longer-term ROI is lower. The stakeholders can then decide whether the longer wait for a higher ROI in Option B is preferable to the shorter wait for a lower ROI in Option A. Option Rapid-B illustrates why stakeholders are interested in rapid application development. If Rapid-B can be developed in half the time, it will be much preferable to either of Options A or original-B.

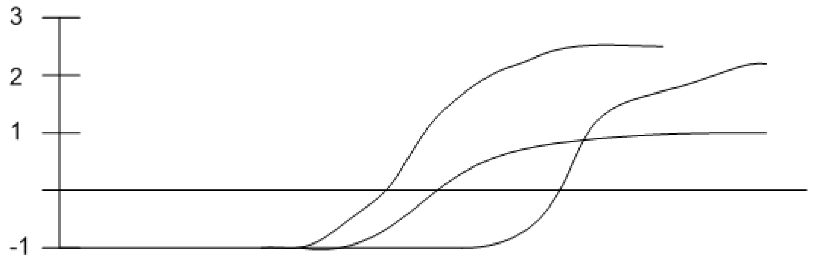


Figure 12 Example of Business Case Analysis Results

Two additional factors may be important in business case analysis. One involves unquantifiable benefits; the other involves uncertainties and risk.

In some cases, Option A might be preferred to Option B or even Rapid-B if it provided additional benefits that may be difficult to quantify, such as controllability, political benefits, or stakeholder good will. These can sometimes be addressed by such techniques as multiple-criterion decision-making or utility functions involving stakeholders' preferences for financial or nonfinancial returns.

In other cases, the benefit flows in Figure 12 may be predicated on uncertain assumptions. They might assume, for example, that the Option B product will be the first of its kind to enter the marketplace and will capture a large market share. However, if two similar products enter the marketplace first, then the payoff for Option B may be even less than that for Option A.

If the profitability of early competitor marketplace entry can be quantified, it can then be used to determine the relative value of the rapid development Option Rapid-B. This value can then be used to determine the advisability of adopting practices that shorten schedule at some additional cost. An example is pair programming: empirical studies indicate that paired programmers will develop software in 60-70% of the calendar time required for an individual programmer, but thereby requiring 120-140% of the cost of the individual programmer. If the profitability of early competitor marketplace entry is unknown, this means that making a decision between the cheaper Option B and the faster Option Rapid-B involves considerable uncertainty and risk. It also means that there is a value in performing competitor analysis to determine the probability of early competitor marketplace entry, or of buying information to reduce risk. This kind of value-of-information analysis can be performed via statistical decision theory.

4. Continuous Risk and Opportunity Management

Risk analysis and risk management are not just early business case analysis techniques; they pervade the entire information system life cycle. Risk analysis also reintroduces the people factor into economic decision-making. Different people may be more or less risk-averse, and will make different decisions in similar situations, particularly when confronted with an uncertain mix of positive and negative outcomes.

A current highly-debated issue is the use of plan-driven methods versus use of agile methods such as Extreme Programming, Crystal Methods, Adaptive Software Development, and Scrum. Recent workshop results involving plan-driven and agile methods experts have indicated that hybrid plandriven/ methods are feasible, and that risk analysis can be used to determine how much planning or agility is enough for a given situation.

5. Concurrent System and Software Engineering

The increasing pace of change in the information technology marketplace is driving organizations toward increasing levels of agility in their software development methods, while their products and services are concurrently becoming more and more software-intensive. These trends also mean that the traditional sequential approach to software development, in which systems engineers determined software requirements and passed them to software engineers for development, is increasingly risky to use. Increasingly, then, it is much more preferable to have systems engineers and software engineers concurrently engineering the product's or service's operational concept, requirements, architecture, life cycle plans and key sections of code. Concurrent engineering is also preferable when system requirements are more emergent from usage or prototyping than prespecifiable. It is further preferable when the relative costs, benefits, and risks of commercial-off-the-shelf (COTS) software or outsourcing decisions will simultaneously affect requirements, architectures, code, plans, costs, and schedules. It is also essential in determining cost-value tradeoff relationships in developing software product lines.

6. Value-Based Monitoring and Control

A technique often used to implement project monitoring and control functions in the software CMM or the CMMI is Earned Value Management. The Earned Value Management process is generally good for tracking whether the project is meeting its original plan. However, it becomes difficult to administer if the project's plan changes rapidly. More significantly, it has absolutely nothing to say about the actual value being earned for the organization by the project's results. A project can be tremendously successful with respect to its cost-oriented "earned value," but an absolute disaster in terms of actual organizational value earned. This frequently happens when the resulting product has flaws with respect to user acceptability, operational cost-effectiveness, or timely market entry.

Thus, it would be preferable to have techniques which support monitoring and control of the actual value to be earned by the project's results.

7. Change as Opportunity

Expending resources to adapt to change is frequently treated as a negative factor to avoid. Software change tracking systems often treat changes as defects in the original requirements. Quality cost systems often treat change adaptations as a quality cost to be minimized. These criteria tend to push projects and organizations toward change-aversion.

Nowadays, changes are continually going on in technology, in the marketplace, in organizations, and in stakeholders' value propositions and priorities. And the rate of change is increasing. Organizations that can adapt to change more rapidly than their competition will succeed better at their mission or in the marketplace. Thus the ability to adapt to change has business value. And software is the premier technology for adaptation to change. It can be organized to make the cost of changes small as compared to hardware. It can be updated electronically, in ways that preserve continuity of service as the change is being made.

Thus, change as opportunity for competitive success is a key economic and architectural driver for software projects and organizations.

Further explanations and examples for Value-based Software Engineering and the VBSE key elements can be found in [12] and [10].

2.4 Application Lifecycle Management

According to [120] Application Lifecycle Management (ALM) can be defined as follows:

- ALM is a discipline, as well as a product category. With so many vendors talking about ALM, it’s sometimes hard to remember that it can be accomplished without supporting tools. Each of the three pillars of ALM — traceability, process automation, and reporting and analytics — corresponds to a manual process that can be made more efficient and effective through tool integration. For example, one bank told us: “Tracing from use cases to test cases manually using spreadsheets is very labor-intensive. That’s why we’re looking for a requirement management tool to integrate with our test management tool.”
- ALM doesn’t support specific life-cycle activities; rather, it keeps them all in sync. A development effort can still fail miserably even if analysts document business requirements perfectly, architects build flawless models, developers write defect-free code, and testers execute thousands of tests. ALM ensures the coordination of these activities, which keeps practitioners’ efforts directed at delivering applications that meet business needs.
- An ALM solution is the integration of life-cycle tools, not merely a collection thereof. Effective tool support for ALM connects the practitioner tools within a development project, such as an IDE, a build management tool, and a test management tool. It’s the connections, rather than the tools themselves, that make up an ALM solution. As an IT exec at one multichannel retailer put it: “You have to pick tools, obviously. But tools are not the focus; the focus is how the tools connect.”

Application Lifecycle Management aims at supporting [120], [119]:

- **Traceability of relationships between artifacts**. Correlating life-cycle artifacts like requirements, models, source code, build scripts, and test cases helps demonstrate that the software has delivered functions as the business wanted it to. Furthermore, internal and external compliance requirements, as well as the increasing need to coordinate development across roles, locations, and organizations, make traceability more of a necessity than an ideal. For most organizations, traceability is a manual process. The problem isn’t just the size of projects; it’s also the number, the varying size and scope, and the artifact interdependencies. Managing dependencies between high-priority change requests and ongoing application development efforts “sometimes seems like it isn’t humanly possible,” reports one healthcare company.
- **Automation of high-level processes**. Development organizations commonly employ paper-based approval processes to control handoffs between functions like analysis and design or build and testing. ALM improves efficiency by automating these handoffs and storing all associated documentation. One financial services firm I spoke with estimated that automating of builddeploy-test processes would save each of its developers an hour a day. Executable process descriptions — process models that correspond to actual automated processes — are a real boon for the many shops that have a “Book of Process” that sits on the shelf and is largely ignored. As one firm put it: “We had a consulting company define a methodology for us. We still have it on a shelf somewhere. A process needs to live in the tools we use if it’s ever going to be followed.”
- **Providing visibility into the progress of development efforts**. Most managers have limited visibility into the progress of development projects; what visibility they have is typically gleaned from subjective testimonials rather than from objective data. A bank we

spoke with told us: “We do progress reporting the same way we’ve been doing it for 40 years. It’s all manual: weekly status meetings, progress reports, demonstrations. We’d love to get test results from nightly builds posted somewhere instead of having to run people down to ask them whether things are working yet.”

2.5 Chapter Summary

Global software development is the chosen context of my research work. Many dependency management approaches in this context are requirements tracing approaches. Thus, requirements engineering is an important fundament of my research work. I apply value-based software engineering and application lifecycle management principles to improve dependency management in Global Software Development projects.

After outlining the major fundaments of this work, the following chapter summarizes the basic research methods I applied as prerequisite for my actual research contributions.

3 RESEARCH METHODS

This section describes basic research methods that I used to reach fundamental conditions for the research contributions described in the following chapters. These basic research methods are (a) a systematic literature review in the field of dependency management, and (b) model building and evaluation, and (c) empirical case studies.

Regarding (a), each research work is based on and refers to existing research approaches. One of the first tasks when starting a research work is to review existing research contributions. A systematic literature review is a good means to identify relevant research contributions in a comprehensible and reproducible way. Thus, in [109] we performed a systematic literature review for dependency management, more precisely for requirements tracing approaches. Section 3.1 describes how we proceeded with the systematic literature review and reports the results.

Furthermore, I developed models consisting of factors that influence dependency management and evaluated my approaches; mainly using case studies. Thus, I shortly outline how to build models and to use case studies for evaluation.

3.1 Systematic Literature Review

Literature surveys have proved to be a very powerful instrument for scientific work. They aim to make implicit knowledge explicit and offer a "big picture". They also offer the broad view on how and where further research on the topic is necessary.

"A systematic review is a means of identifying, evaluating, and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest." [79]

Coming from the domain of medical research literature reviews originally tried to combine the results of several clinical studies (primary studies) into a single consistent work (secondary study). In the domain of medical research these studies are much more structured and there exist several practices to compare and assess quantitative results. Quantitative reviews (QNR) aim to synthesise empirical findings of studies in the same field of research. They are mostly about comparing data and study results and weighing them up against each other. Qualitative reviews (QLR), in comparison, also try to compile case studies, experience reports etc. to a single consistent work. They do not neglect empirical results but focus on the very conclusion of the study instead of detailed numerical results.

Simple vs. systematic reviews

A simple review does not emphasize the methodology of the review itself. In comparison, the systematic review contains a specific pattern of sections such as background, purpose, data sources, and study selection. Furthermore, it presents specific requirements on the review which were defined apriori.[11]

However, in the domain of computer science studies differ very much in their focus and methods. Thus, I used a lightweight process to conduct systematic reviews in the field of software engineering. Our process is a summary of many works in the field of systematic reviews which include [11][15][16] and [79].

Our process is specially tailored to compare and synthesise qualitative results as Brereton et al. report in [16] that software engineering systematic reviews are likely to be qualitative in nature.

3.1.1 The Systematic Literature Review Process

A systematic review differs from a traditional review in being formally planned and executed. It is therefore replicable and readers can understand how the authors came to their findings.

Despite these advantages I have to admit that many of the authors, publishing about systematic reviews, state that it takes “considerably more effort” than a traditional review. This should be kept in mind when deciding to apply this method.

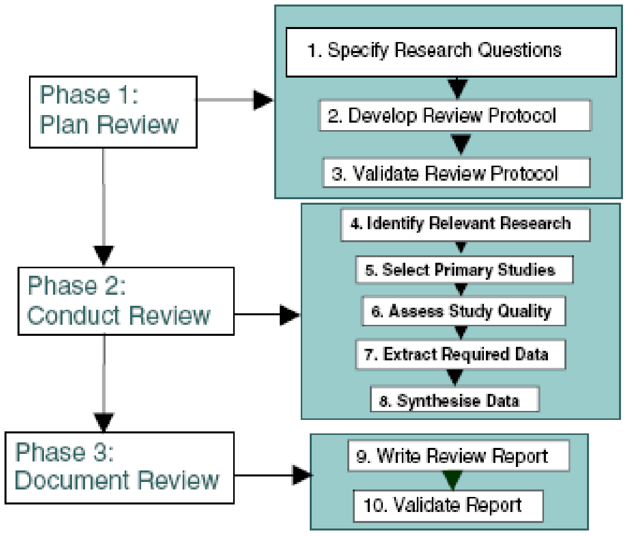


Figure 13 Phases of a Systematic Literature Review

The basic process, which was developed by Kitchenham et al. [79], is depicted in Figure 13: From the three phases planning, conducting and reporting the review described in the figure, I want to focus on the planning phase, because the major decisions are made at this time. I want to propose an easy-to-use template for writing the review protocol (a document that describes the setting of the systematic review) and show which process steps are necessary to get there.

Writing the Review Protocol

The review protocol heavily steers and influences the systematic review process. Therefore it is worth spending some effort in developing and validating it to minimize the risk of bad surprises during the review process itself. The phases of the systematic review protocol are described in detail below. The following Figure 14 shows in which order they are conducted and what the output of the phases is.

Our process is a little different form the original works by Kitchenham et al. In [79] it is proposed to validate each phase. I assume that writing the review protocol is a single process and the document is written at once. With multiple researchers carrying out the systematic review process, discussions offer some quality assurance. Sole researchers should consult external experts (actor on the left hand side) to get a suitable initial review protocol. From Woodall et al. we get the information that short iterations of feedback from their supervisor are essential for sole researchers.

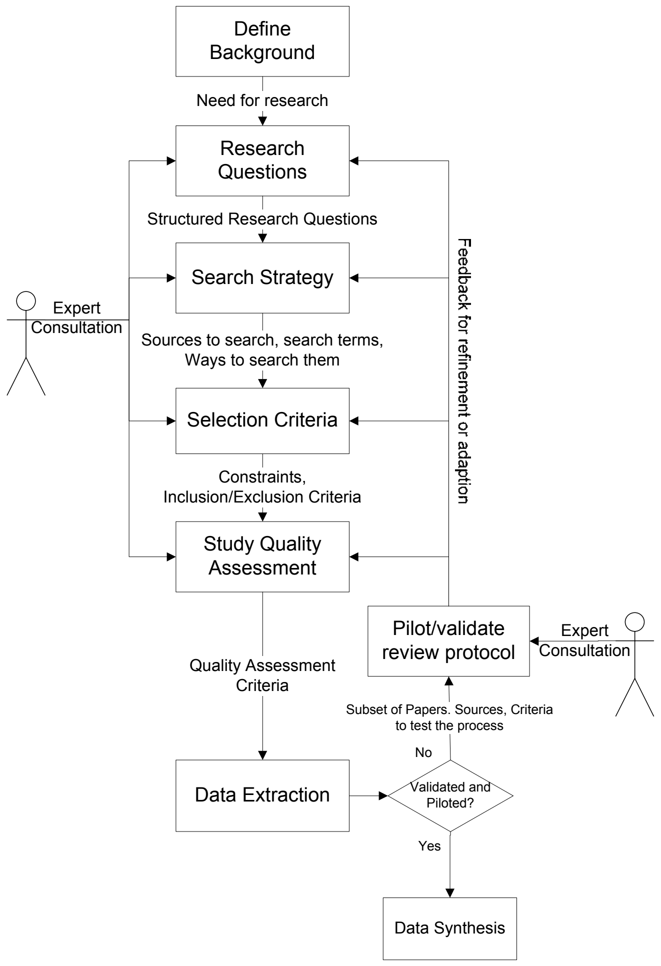

Define Background
Need for research
Research Questions
Structured Research Questions
Search Strategy
Sources to search, search terms, Ways to search them
Selection Criteria
Constraints, Inclusion/Exclusion Criteria
Study Quality Assessment
Quality Assessment Criteria
Data Extraction
Expert Consultation
Feedback for refinement or adaption
Pilot/validate review protocol
Expert Consultation
Subset of Papers. Sources, Criteria to test the process
No
Validated and Piloted?
Yes
Data Synthesis

Figure 14 The process of writing and piloting the review protocol

Background

The background section provides the rationale for the survey. The topic should be motivated in general. Possible motivations could be that there is no literature survey available on the topic or the existing ones are out of date.

Research Questions

The research questions are the major part of the systematic review and their specification. It is the most critical element of a systematic review [15]. Formulating research questions requires some background knowledge about the topic; an overview paper or a book on the topic can help to get this.

First it is recommendable to formulate questions in natural language. The systematic review offers several aspects of a question, which enable us to decompose the question and identify possible weaknesses in the questions. There is a very elaborate template for the question formalization by Biolchini et al. in [11]. However, these templates come from the field of quantitative systematic reviews and some points are optional for qualitative reviews. I marked those points with an asterisk.

- **Question Focus**: defines the systematic review focus of interest, i.e. the review research objectives. Here, the researcher must decide what he/she expects to be answered in the end of the systematic review.

- Question Quality and Amplitude
    - **Problem**: defines the systematic review target, describing briefly the research context.
    - **Question**: research question to be answered by the systematic review. It is
    - **Keywords and Synonyms***: list of the main terms that compose the research question. These terms will be used during the review execution (in case the search by keywords is chosen as study selection methodology). In some reviews the keywords are the same for every question and can also be defined only in the section search strategy.
    - **Intervention**: what is going to be observed in the context of the planned systematic review?
    - **Control***: baseline of initial data set the researcher already possesses. For qualitative reviews e.g. the validation of a concept, this concept can be seen as a Control as well.
    - **Effect**: type of results expected in the end of the systematic review
    - **Outcome Measure***: metrics used to measure the effect
    - **Population***: population group that will be observed by the intervention.
    - **Application**: defines the target audience for the answers of the research questions

Experimental Design*: describes how meta-analysis will be conducted, defining which statistical analysis methods will be applied on the collected data to interpret the results.

So this template helps us to get clearance on scope, target groups etc. of our research question if necessary. Another interesting possibility to properly scope research questions can be to formulate research questions which are explicitly NOT subject of the systematic review.

Traditionally, research questions are rather hypothesis-centered [16], but often they also just aim to "identify research issues". This is especially true for state of the art reviews. It should be clear of which kind your specific questions are.

Finally research questions should not be seen as fixed when they have been once formulated. It is very likely that the research questions change after having piloted the review protocol. It is also possible that two rather general questions are later decomposed into four or five detailed questions.

Search Strategy

The search strategy defines which sources will be searched and via which way they will be accessed, e.g. are conference proceedings retrieved via the library or on-line? For each source it must be replicable how and why the search led to the published results. So the following information must be provided.

- Name of source
- Years/Issues searched (can be defined for all sources at one time)
- Date of search (can be of coarse granularity e.g. months)

Furthermore it is beneficial to group the sources by "conference", "journal" and other sources. To find all relevant sources the list of sources should be validated with a researcher who has already worked in this field, he/she can help in completing the list, so no relevant sources are omitted.

Selection Criteria

Search Terms

These criteria define which material obtained from the sources above will be included or excluded into/from the review. A major part is the search terms used. These can be stated just as a simple list or with sophisticated regular expressions. Where do you look for the search terms, in the title, in the abstract in the keywords or in the full text?

It is possible to structure search terms with sophisticated regular expressions, but the problem is that different search engines interpret these terms in a different way [16]. So it is better to state them in normal text and to explain in natural language for which combination of terms you will search.

Restrictions

Some general restrictions on the search should also be told. For example which kind of material you will use i.e. whether you are just using papers which are available for free or are you using material you have to pay for as well. Furthermore, it should also be defined in which languages we are searching for materials, this clearly affects the search terms. In the domain of computer science English is the dominating language, though it might be valuable in some cases to scan others if you have the skills to.

For documenting the search results a spreadsheet is useful, especially if its fields already correspond in some way with the Data Extraction Form that I will design later. Such a spreadsheet for recording search results makes it easy to keep track of the actual status of the literature review. It is recommendable to keep the fields of the spreadsheet similar to those of the data extraction form I will develop later, so data can be easily exchanged between the overview and the detailed view of the review. An example of the overview spreadsheet can be found in the appendix section 9.1. Often these criteria are already implicitly defined, so it is not much effort to write them down with a few sentences.

Finally it must be traceable how the search led to the result set of publications that are finally used for the review.

Study Quality Assessment

An important decision is when a specific publication is relevant for your review or not. Here inclusion and exclusion criteria have to be defined. Multiple researchers should discuss these criteria and come to an agreement. A single researcher should consult an external expert to validate these questions.

Furthermore, there exists a detailed classification to assess the quality of empirical evidence by Kitchenham [79], as depicted in Table 1.

Kitchenham also states that such classifications are not always applicable, as the nature of software engineering experiments is very different from the domain of medical experiments. So this should serve just as a general guide in which context an experiment stands.

For qualitative reviews such a classification is not available. Often case studies and concept papers account for a great amount of the retrieved literature. They are often biased by the author's opinion. So I have to examine different attributes of the paper to assess its quality. A few questions can help in finding these attributes:

- Is the paper published or "grey literature"?
- Where was the paper published? Conference, Journal etc.?

- How often was the paper cited? (May not be possible to say for very new papers)

Table 1 Criteria to assess the quality of empirical evidence

Rating	Type of Evidence
1	Evidence obtained from at least one properly-designed randomised controlled trial
2	Evidence obtained from well-designed pseudo-randomised controlled trials (i.e. non-random allocation to treatment)
3-1	Evidence obtained from comparative studies with concurrent controls and allocation not randomised, cohort studies, case-control studies or interrupted time series with a control group
3-2	Evidence obtained from comparative studies with historical control, two or more single arm studies, or interrupted time series without a parallel control group
4-1	Evidence obtained from a randomised experiment performed in an artificial setting
4-2	Evidence obtained from case series, either post-test or pre-test/post-test
4-3	Evidence obtained from a quasi-random experiment performed in an artificial setting
5	Evidence obtained from expert opinion based on theory or consensus

Furthermore, there are expert-authors for every specific topic, who publish often and successfully in their field. A researcher who has already worked in the specific field can help in identifying them, when you are new to the topic. Their works can serve as a baseline for further research.

Reading the abstract of the paper is a commonly accepted way of assessing the quality or relevance of a paper throughout different fields of science. But from [16] and [79] we learn that the quality of abstracts in software engineering is comparably low. They therefore suggest also reviewing the conclusions of the papers.

Finally it is recommendable not just to delete studies, which drop out of the list of relevant papers according to the inclusion or exclusion criteria, but also to keep a list of excluded sources.

Data Extraction Process

According to the research questions and corresponding search terms, the full papers are acquired. From online resource we often get the full papers with no meaningful filenames. So it is very recommendable to rename them according to a standardised convention. Such a filename convention could look like the following:

Author.et.al_TitleOfPublication_YYYY.<file extension>

YYYY stands for the year of the publication and the file extension will be Portable Document Format (.pdf) most of the times.

This enables us to discover papers by simply looking at the file name as well and we do not need an additional list. We choose not to use spaces or special characters like “/”, “\”, “&” etc. in the filename as this could lead to potential problems if our retrieved work would be inserted in a web-based system.

The next objective of this stage is to accurately record the information researchers need for their work. For this a so-called Data Extraction Form should be designed. Besides recording information about the paper like author title year etc. it should provide the researcher with a comfortable structure to collect quotes for his/her research questions. Another approach could be to insert check boxes or text fields if we want to extract very limited information. This could be the case if one research question requires us just to see how many papers have a specific attribute.

In the field of quantitative reviews it is sometimes necessary to manipulate extracted data to make them comparable. If this is the case the process and the corresponding actions should also be documented in this section.

There should also be a section to record off-question information to be not too restricted. Such a Data Extraction Form could look like the one presented in the appendix section 9.2.

Advantages of the Data Extraction Form:

- Extracted information does not disappear as a little citation in a large paper and it is traceable which information from a specific paper was sensed relevant by a specific researcher.
- It facilitates the collaboration of multiple authors, as filled out forms can be forwarded and processed by other persons.

Disadvantages of the Data Extraction Form:

- Filling out the data extraction form causes additional effort.
- Extracted information is scattered over several forms at the beginning

Data extraction means collecting data and quotes from the primary study to include them in the secondary study.

The idea is to have all relevant information of a publication in the Data Extraction From so that there is no more need to deal with the source paper any more. Therefore time consuming searches in 30-page long papers should not be necessary. Furthermore it is still traceable on base of the single papers how the researchers dealt with the single papers i.e. what they perceived to be important.

Data Synthesis

Data Synthesis is not really a part of the review protocol, but it is recommendable to think about this step in the planning phase as well.

Initially, data synthesis was the task of combining and normalizing statistical data collected from the primary studies.

Clearly the focus should be on answering the research questions, as they are the reason to conduct the review, but is also beneficial to have a basic structure of the paper. With this we are able to compose a consistent work with introduction and background information.

3.1.2 Piloting/Validating the Protocol

This protocol should now be piloted with a few search terms specified in the protocol. Keeping the inclusion/exclusion criteria in mind a small initial set of publications should be acquired. Next, the quality assessment criteria should be applied to the publications and data extraction should be tried. Shortcomings detected during this pilot-phase give direct feedback on the respective phase of the review protocol, e.g. the search questions could lead to no relevant material, or the research questions might state the wrong scope on the topic.

A short validation for small reviews could also be to have the review protocol checked by an experienced researcher in the respective field.

Having incorporated the feedback from the pilot phase I am ready to apply the protocol to the whole set of publications and to move to the next phase "Conducting the Review".

3.1.3 Conducting the Review

Conducting the review is mainly carrying out what has been specified in the review protocol. The following subsections shortly describe which additional activities have to be done in the corresponding phases of the review.

Estimation of effort

Biolchini et al. give an average hour effort, calculated from four systematic reviews in the field of software engineering. Effort for the following phases was:

Table 2 Estimation of efforts

Phase	Effort	Relative Effort
Planning	11hrs	13,9%
Evaluation:	1hr	1,2%
Creating Versions:	4hrs	5%
Studies Search:	31hrs	39,2%
Studies Evaluation:	32hrs	40,5%

Note that the phases "data extraction" and data synthesis are not included in this compilation.

Identification of research

This phase includes searching the specified sources for the respective search terms. It can be beneficial to record information about the search and its results, especially when using search engines. Such information could be:

- Source searched
- Search term used
- Date of the search
- Results (Author, Title, Volume, Issue, Pages Year)

Selection of primary studies

In this phase the abstracts are examined and the relevance is judged. Inclusion and exclusion criteria from the review protocol are applied as well.

Study quality assessment

If specified, the primary studies produced in the previous stage, are now evaluated according to the study quality assessment criteria.

Data extraction

Reading and filling out the Data Extraction Forms can be performed independently by multiple researchers, but some of them should be compared to reach consensus about how the data extraction process should be carried out. Another possibility of quality assurance could be to let one researcher extract the information and the other one just checks his work. According to [16] there is no major difference in quality between these two methods. Furthermore, it can be validated by an external expert. Another method of ensuring consistency is to let all researchers review some identical papers and to check whether they extract data in a consistent manner.

For single researchers such a possibility is obviously not available; here the supervisor of the paper could also review one paper and the student can compare his findings with those of the supervisor.

When working in the field of quantitative reviews data extraction might also involve normalising and manipulating data to make them comparable.

Data synthesis

Having retrieved all the relevant information I can start to synthesise the relevant information. For advanced quantitative reviews this involves statistical methods for meta-analysis which are beyond the scope of this paper. Approaches for this can be found in [79].

Despite the potential that meta-analyses provide, I learn from other sources such as [16] that it might not even be possible at all in the domain of software engineering. This is because reporting protocols vary so much from study to study.

In the following Kitchenham [79] states how non-quantitative information should be summarised:

- Extracted information about the studies (i.e. intervention, population, context, sample sizes, outcomes, and study quality) should be tabulated in a manner consistent with the review question.
- Tables should be structured to highlight similarities and difference between study outcomes.
- It is important to identify whether results from studies are consistent one with another (i.e. homogeneous) or inconsistent (e.g. heterogeneous).
- Results may be tabulated to display the impact of potential sources of heterogeneity, e.g. study type, study quality, and sample size.

Even if we often cannot directly compare results or findings in the domain of software engineering another possibility is to count positive or negative results on a specific practice as “votes” for or against it. These votes should also be supported by the data extraction form, which can be found in the appendix section.

3.1.4 Reporting the Review

For qualitative reviews data synthesis and reporting the review might be one step. They are separated in our structure, because if I employ meta-analysis, data synthesis is concerned with calculations and therefore reporting is the task of presenting these results in a consistent work.

In [79] I also find the task of validating the review. This can be achieved through an external researcher/expert or via peer review of the paper.

3.1.5 The Review Protocol for a Systematic Review on Requirements Tracing Approaches

This subsection describes how I applied the systematic literature review process to the field of dependency management (with focus on requirements tracing approaches) and how the concrete instance of the review protocol looks like. The review was performed in May 2007.

I defined four tracing activities as prerequisite for our survey. They are taken from our Tracing Activity Framework (TAF) that is described in more detail in section 4.1:

Trace Specification is the activity where the project manager defines the types of traces that the project team should capture and maintain. For example, the project manager can decide to capture traces between requirements, source code elements, and test cases.
Trace Generation is the activity of identifying and explicitly capturing traces between artifacts. Methods for trace generation range from manually capturing traces in matrices or requirements management tools that automatically create traces between artifacts based.
Trace Deterioration is more the impact of external events than an activity. Traces can degrade over time as related artifacts change. If only the artifacts are updated, e.g., due to change requests, and the traceability information is not updated, the set of existing traces is likely to get less valid over time. Deterioration of traces affects the value of traces, because it reduces the correctness and completeness of traces.
Trace Validation and Rework. Trace validation is the activity that checks if the existing traceability information is valid or needs to be updated, e.g., identify missing trace links. I call the updating of traces "trace rework". Trace validation and trace rework are often performed together as they ensure correct and up-to-date traces and counter trace deterioration effects.

Background
Requirements Engineering is a key factor whether a software system is successful or not. Even when perfectly programmed, if the system implements the wrong requirements it will be a failure. But even if the right requirements have been found it remains a great challenge to manage them. Frequent changes, documentation issues and collaboration between multiple stakeholders bring complexities to requirements management. This is where requirements traceability enters the scene. Traceability of requirements offers many benefits for medium to large size development projects. Among them is the analysis of change impact, cost estimation of changes or tracing to a specific author of a requirement. Though there are many publications on this topic, it is surprising, that there exist still very few literature surveys on the topic of requirements tracing today. The ones which are available are either old or do not focus on a procedural model for requirements tracing. With this work I therefore aim to fill this gap.

Research Questions
RQ: Which activities can be identified in the tracing process, i.e. do the activities mentioned in literature match with our defined tracing activities?

- Question Focus: To identify activities in the process of requirements tracing. Validate the activities in the tracing activity framework.
- Question Quality and Amplitude
    - Problem: Much research has been conducted on requirements tracing. But form a process view results are still scattered and unordered, there are many isolated works. The tracing activity model aims to standardize activities in the tracing process, but is must be verified whether the TAF models the reality.
    - Question: Does the TAF model activities that are actually referred to in literature?
    - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking
    - Intervention: publications that deal explicitly with requirements tracing, mentioning of activities and process models for requirements tracing.

- Control: Tracing Activity Framework
- Effect: Works about requirements
- Outcome Measure: Qualitative results: how to deal with the tracing activities, naming conventions on the activities, potential additional activities. Quantitative results: how many mentions of which tracing activities.
- Population: 100+ Publications regarding software engineering, requirements engineering.
- Application: Researchers in the field of requirements engineering, software project managers
- Experimental Design: no statistical method is going to be applied.

RQ: Which tracing parameters exist in literature and how can they be mapped to TAF?

- Question Focus: To identify measures and factors which are connected with requirements tracing
- Question Quality and Amplitude
  - Problem: In the numerous works on requirements tracing, many different quantitative and qualitative measures are mentioned. However many of them are ambiguous and sometimes one measure is even used to describe two different ideas.
  - Question: Which parameters are used to describe requirements tracing and which factors influence it.
  - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking, software metrics (limited set of papers only for disambiguation)
  - Intervention: publications that deal explicitly with requirements tracing, where requirements activities are described with any kind of specific measures. Reference publications in the field of software metrics
  - Control: Common metrics of software engineering (e.g. lines of code…) Metrics of the Tracing Activity Model.
  - Effect: Set of all measures used in the field of requirements tracing enriched with explanations and examples. Furthermore the measures should be reasonably partitioned and linked to the Tracing Activity Model to see which measures influence which activity.
  - Outcome Measure: Number and quality of the measures found.
  - Population: 100+ Publications regarding software engineering, requirements engineering.
  - Application: Researchers in the field of empirical software engineering and requirements engineering, software project managers
  - Experimental Design: Valuing the measures in connection with the importance of the publication where they are mentioned (counting the mentions of measures).

RQ: What methods and tools exist to conduct a tracing activity?

- Question Focus: Carrying out the respective tracing activities
- Question Quality and Amplitude
  - Problem: Filling out traceability matrices can be a very mundane and cumbersome duty, especially in large scale projects. Therefore some approaches have been

developed to facilitate this process. However leverage traceability information it must be possible to retrieve it again.
    - Question: What approaches exist to ease the activities of requirements tracing?
    - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking
    - Intervention: publications that deal explicitly with requirements tracing
    - Control: Information retrieval metrics found and defined in the research question above.
    - Effect: Conclusion which methods and tools are feasible to conduct a specific trace activity.
    - Outcome Measure: quantitative comparison of the different information retrieval metrics, experience reports in papers.
    - Population: 100+ Publications regarding software engineering, requirements engineering.
    - Application: Researchers in the field of empirical software engineering, requirements. Practitioners willing to introduce or improve requirements tracing

RQ: What are the effects of requirements tracing approaches reported in literature in terms of precision, false positives, and utility for trace applications?

- Question Focus: Effects of RT approaches.
- Question Quality and Amplitude
    - Problem: Justifying RT is often a difficult task. Benefits of traceability are not summarized and explained in a compact way
    - Question: What are the effects of RT approaches reported in literature?
    - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking
    - Intervention: publications that deal explicitly with requirements tracing
    - Control: Precision, false positives, utility for trace applications
    - Effect: Summary to which effects different RT approaches lead.
    - Outcome Measure: quantitative comparison of the different information retrieval metrics, experience reports in papers.
    - Population: 100+ Publications regarding software engineering, requirements engineering.
    - Application: Researchers in the field of empirical software engineering, requirements engineering and information retrieval, software project managers

RQ: How do tracing parameters influence the tracing activities?

- Question Focus: Effects of RT approaches.
- Question Quality and Amplitude
    - Problem: Parameters for requirements tracing are very numerous, their influence on each other is only partially reported in some papers. Furthermore we do not know in which phase (activity) of the tracing process, which respective parameter is important.
    - Question: Relationships between activities, parameters, and effects?
    - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking

- Intervention: publications that deal explicitly with requirements tracing
- Control: Mentioned relationships between parameters in obtained literature.
- Effect: Conclusion which parameters influence each other and which parameters can be mapped to specific tracing activities.
- Outcome Measure: References in the description of a parameter to the related ones, graph showing these relationships.
- Population: 100+ Publications regarding software engineering, requirements engineering.
- Application: Researchers in the field of empirical software engineering, requirements engineering and information retrieval, software project managers

RQ: What are open issues of requirements tracing research that have not yet been covered?

- Question Focus: Effects of RT approaches.
- Question Quality and Amplitude
    - Problem: As there is no encompassing survey on RT there also does not exist an overview which subtopics might need more attention in the future
    - Question: Open Issues in RT for research that has not yet been covered?
    - Keywords and Synonyms: requirement, tracing, traceability, tracking, linking
    - Intervention: publications that deal explicitly with requirements tracing
    - Control: Topics already covered by traceability research
    - Effect: Summary to which topics are still uncovered.
    - Outcome Measure:
    - Population: 100+ Publications regarding software engineering, requirements engineering.
    - Application: Researchers in the field of empirical software engineering, requirements engineering and information retrieval, software project managers

Search Strategy
Firstly I scan proceedings of the most important scientific conferences on the topic. These include:

Conferences
- IEEE Int. Conf. on Requirements Engineering
- (IEEE/ACM) International Conference on Automated Software Engineering (former KBSE)
- International Conference on Software Engineering (ICSE)
- European Software Engineering Conference (ESEC)

Journals
- Transactions on Software Engineering (ACM)
- ACM Trans. on SE Methodology (TOSEM)
- Journal on Systems and Software
- Requirements Engineering Journal
- IEEE Software
- IEEE Computer

For these sources I will also follow their references if they seem relevant for our topic.

Here I hope to find contributions that were not covered by our subset of conferences, but are valuable for the topic as well. For each search conducted with one of these engines, I will record the date of search, the search term, the full result list and our subset of papers I added to the selection.

All our searches have been conducted between 02/2007 and 04/2007

Our research dates back to the year 1992 because I think that with last 15 years the major part of the evolvement of requirements tracing is covered (as requirements tracing is a rather young topic of software engineering).

Selection Criteria

On one hand I scan whether “requirements tracing” “traceability” or “trace” are in the title, the keywords or the abstract of the paper. Papers which deal with traceability but do not have this attributes are excluded. On the other hand I look for the topic-categories of the conferences. Clearly our focus also lies on the traceability category. Detailed selection criteria can be found in the corresponding research questions.

The study will only include papers which are available at no charge, or via electronic libraries where the Vienna University of Technology is subscribed to.

Study Quality Assessment Procedure

An important quality measure is the number of citations for a specific paper and its age. So new, frequently cited papers are the most important ones, while old rarely cited ones are of lower priority. The number of citations stated by Google-Scholar will provide us with the necessary information. Being aware of the fact that this information may only cover recent publications in some cases, we will cross check citation information with other services, such as CiteSeer if necessary. Moreover not the exact number of citations is relevant for our assessment, but the magnitude (e.g. 1, 10, 50 citations) determines the importance of the publication. It is obvious that for very new publications citation information will not be available. Here I have to assess the quality by scanning the paper or based on the author or the way of publication. Furthermore, I rate journal papers over conference papers.

As I am not primarily conducting a survey just on empirical data I am not able to state detailed inclusion or exclusion criteria by numbers. Also I am aware of the fact that it is likely to find only positive results in the studies, as negative often do not get published. I will try to find some “grey literature” as well, but for this I cannot claim full coverage.

Data Extraction Process

According to our research questions I acquire the relevant full papers. These papers are stored in a folder by the name of the publication e.g. “RE Conference”. I rename the works by the schema:

Author.et.al_TitleOfPublication_YYYY.<file extension>

YYYY stands for the year of the publication and the file extension will be preferably Portable Document Format (.pdf)

This enables us discover papers by simply looking at the file name as well and I do not need an additional list. I choose not to use spaces or special characters like “/”, “\”, “&” etc. in the filename as this could lead to potential problems if our retrieved work would be inserted in a web-based system.

In the first iteration I read the abstracts of the papers and try to rate their relevance. Secondly I rate the papers by relevance and quality by reading the abstract and the conclusion. Here I have only a binary relevance measure include or discard.

Data Extraction Documentation

On one hand things as the measures, or approaches for activities, will be recorded in a spread sheet for all the papers. This allows traceability where which attribute was found.

On the other hand I will do a short review with our Data Extraction Form (see template attached) for each paper I write down the structure and relevant concepts regarding our research questions. The form is created using MS Word, I store it in the same folder as the reviewed publication. I also use the same filename convention as with the publication to name the Data Extraction Form, so when sorting the folder by name I see which files are connected and I can distinguish between paper and review simply by the file extension.

In the Data Extraction Form I primarily want to record qualitative information. Quantitative information for example whether the paper has a certain attribute or not should finally reside in the spreadsheet. Of course this creates some additional effort because I always have to maintain two reporting documents, but I hope to save time at the end of the process as it will be easier to report for example the number of mentions or the number of empirical studies.

Data Synthesis

Once I gathered all required information I will compile it in a consistent single work. The main focus will lie on relating the results to the Tracing Activity Framework, to see whether it represents the reality reported by numerous researchers. The structure of the report section will be partitioned by means of the tracing activities, as described in the following subsection.

3.1.6 Results of the Systematic Review on Requirements Tracing Approaches

This subsection reports the results of the systematic literature review on requirements tracing approaches. It is structured according to the tracing activities explained above.

Furthermore, I provide separate subsections for the list of relevant tracing parameters that I identified (these parameters are also assigned to relevant tracing activity subsections) and for trace applications. The following subsections describe the review results.

Trace Specification

In this phase the main decisions for the requirements tracing process are made. From [59] I learn that the project manager defines the types of traces the project team should capture. Such decisions influence the whole outcome of the process. Clearly they are not made "at will" but influenced by certain parameters. These input parameters include: number of requirements, number of artifacts, and volatility of requirements, criticality/risk, project size, complexity, budget, estimated system lifetime, and CMMI level. Input parameters are fixed but in the specification phase I have some other variables with which I can influence and steer the tracing process. These tailoring parameters include: precision/granularity of traces, tracing scope, priority of requirements / traces, number of traces, tool support, automation, and point in time of trace generation. Detailed information about the parameters can be found in section "Parameters for Requirements Tracing".

Trace specification is explicitly mentioned by several sources such as Ramesh et al. in [113] who state that "all the types of traces which must be captured and maintained throughout the project have to be defined." Pinheiro refers to this activity by the term "trace definition"[105][106]. Also Von Knethen identified this activity by "Define Entities and Relationships" in [81]. Heindl and Biffl need a specification phase in their value-based RT-process for packaging of requirements and assigning priorities and trace granularities to them [63]. The goal modelling phase for non-functional requirements mentioned by Cleland-Huang et al. in [22] can also be seen as an implicit mentioning of specification. Finally Arkley and Riddle mention an “Initiation Stage” for their traceable development contract for recording “the aim of the work to be undertaken, a description of the problem artefacts, stakeholders and completion date“[8].

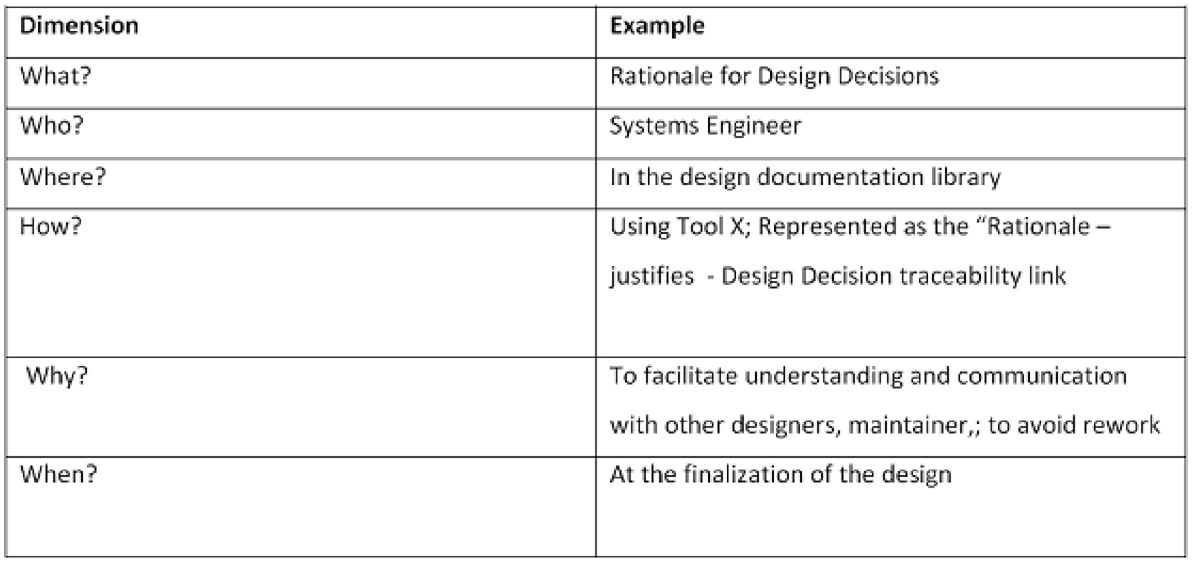
Nevertheless, I can summarise that many researchers do not mention an explicit specification phase at all. They focus on particular trace types and are concerned with the generation of traces right from the start. The purpose of such specifications is twofold: While one group of researchers sees it as a phase to make project decisions, the other group is already concerned with the preparation of requirements to be traced.

Concerning the name of this activity “Trace Definition” seems to be a reasonable term for this activity as well as it would reduce the danger of confusing it with the requirements specification (= requirements document).

Approaches for the specification activity

The following Table 3 introduces some suggestions, which fit very well in the specification phase: A good first step seems to be to reason about the dimensions of requirements tracing [112], which might be helpful in the selection of a tracing approach.

Table 3 Dimensions of Requirements Tracing



Dimension	Example
What?	Rationale for Design Decisions
Who?	Systems Engineer
Where?	In the design documentation library
How?	Using Tool X; Represented as the “Rationale – justifies - Design Decision traceability link
Why?	To facilitate understanding and communication with other designers, maintainer,; to avoid rework
When?	At the finalization of the design

Traceability policy

Traceability information is only valuable if it is permanently maintained and updated. But as this takes much effort, the decision which traceability information to capture and use is very important. This decision results in the traceability policy. Kotonya and Sommerville [83] suggest that the traceability policy should include the following:

- The type of traceability information which should be recorded
- The techniques, such as traceability matrices, which should be used for maintaining traceability
- A description of when the traceability information should be collected during the requirements engineering and system development processes. The roles of the people, such as the traceability manager, who are responsible for maintaining the traceability information, should also be defined.

- A description of how to handle and document policy exceptions, that is, when time constraints make it impossible to implement the normal traceability policy
- The process used to ensure that the traceability information is updated after the change has been made. This should cover both normal and exceptional change processes

Furthermore, the traceability policy is always highly project-specific. For each project a trade-off between completeness and feasibility has to be made. The factors influencing this policy are:

- Number of requirements
- Estimated system lifetime
- Level of organisational maturity
- Project team size and composition
- Type of system
- Specific customer requirements

Trace strategy

Cleland Huang et al. [23] mention the following four strategies and state that: "Without such strategies, individual decisions made during the development process will be unlikely to support the global objectives of the project."

- Maximise the usage of dynamic link generation: Dynamic link generation, through techniques such as IR or heuristic traceability provides an effective strategy for minimizing the number of links that need to be manually established and maintained. The strategy is therefore to replace explicit user-defined links with dynamically generated ones whenever possible.
- Trace for a purpose: In order to minimize negative return traces, it is important to evaluate WHY a link is being created, so that the most appropriate and useful type of link can be deployed. If a potential link serves no clear purpose, then it should not be established.
- Select the most effective traceability mechanism: This strategy proposes that the most effective mechanism be selected for each link. This means that a single requirement could be traced to one artifact using one technique and to another artefact using a different technique (e.g., see section about heterogeneous trace-generation approaches).
- Differentiate between throw-away and long-term traces: This strategy differentiates between traces that are useful only during development, and those that should be maintained in the longterm.

Define a target audience for the traces

I find this concept in the work of Gates et al. [49] "Identification of who will be using trace information and how it is going to be used directs the choice of traceability model. This also governs the complexity with which trace information, comprised of artifacts and relations, is organized and maintained."

Also Ramesh and Jarke mention the differences in how people use traceability information. Management uses traceability to prove compliance with the specification to the customer while project management uses traceability for internal progress and coverage analysis also the generation of Gantt charts from the traceability information is mentioned [112]. Developers on the other hand might use traceability in a much more technical way to browse from specific requirements to software artefacts in their daily work.

Define granularity

Macfarlane and Reilly [90] present the approach of design elements. The artefacts between I want to trace are segmented into design elements. So the design elements of a text document are paragraphs, while the design elements of a source file would be classes, functions, globals etc. The advantage is clear: I can link a specific paragraph of the requirements document to a class or even a method of the source code. This increases the precision/granularity of the tracing process. Certainly this segmentation has to be done in the specification phase, as the artefacts must be parsed to get the design elements. But the parsing operation has to be repeated with any change made to the system.

Prepare for automated trace generation

When employing the technique of automated trace generation it might be important to specify in advance that keywords should be assigned to the design elements. Some automated approaches can make good use of keywords such as those presented in [70].

Furthermore Antoniol et al. also employ their automated approaches under the premise "… that programmers use meaningful names (i.e., names derived from the application and problem domain) for their identifiers…" [4]. On one hand a reasonable assumption, but as I am into the specification and validation of a process model, it might be very recommendable to introduce a small specification phase even for automated approaches. This could include, coding conventions and the introduction of a common project vocabulary. Often such a vocabulary is available as glossary in the project contract. So I could require the programmers to use terms out of this glossary for their work.

Requirement types

Archer [6] rejects highly formal (traceability matrices) and highly informal techniques (user stories). His streamlined approach just uses requirements statements which are one to four sentences long. He distinguishes between business requirement and software requirements. Software requirements aim to fulfil business requirements. He classifies requirements as following:

- Concrete:
    - Primary characteristics: The requirement is linked to a single software deliverable which it specifies and there exists a well defined completion criterion for determining when the deliverable satisfies the requirement;
    - Secondary characteristics: The requirement is linked to: a business requirement, a task in the schedule, a completion status and a software build;
- Immediate: The requirement satisfies the primary characteristics of a concrete requirement, but not the secondary ones;
- Abstract: the requirement is not linked to a software deliverable but applies to every software deliverable in the context of a certain hierarchy.
- Typed: The requirement is relevant for some aspect of the software but does not meet any of the criteria above.

Traceable development contract (TDC)

The traceable development contract is a process-based approach and is a means of controlling the interaction of development teams. Traceability is employed as a means of assessing the impact of a change to the common development artefacts and providing a basis for the negotiation of the change [8]. In the proposed traceable development contract there are four stages:

1. Contract initiation: recording the terms of the contract such as the aim of the work to be undertaken.

2. Problem Discourse: clarify any issues with the problem artifacts and resolve any misunderstandings

3. Proposed Solution: allows a downstream development team to record how their solution will satisfy the problem. Here definition of granularity is undertaken as well.

4. Refinement: Teams are free to make changes to the problem description.

It is interesting to see how Arkley et al. see their traceable development contract within the v-model for software development. They put a strong focus on recoding traceability information between problem and solution.

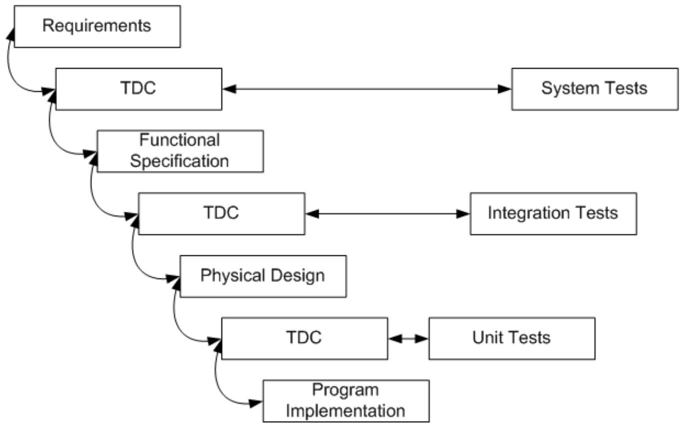


Figure 15 Traceable development contract

The TDC aims to generate traces between requirements and test cases even during the definition of requirements. For example, during the definition of a requirement also the respective test case which validates it is defined and a trace therefore established.

Tracing of non-functional requirements

An approach for tracing non-functional requirements is proposed by Gross and Yu in [53]. They make use of soft goal models and design patterns to support traceability to non-functional requirements. A similar concept is also employed by Cleland-Huang et al. [22]. They also report "Goal Modelling" with soft goals to support requirements tracing for non-functional requirements. They take the so called Softgoal Interdependecy Graph (SIG) as a basis for their solution. An example for such a graph can be found in the section "Dependency between parameters".

Requirements traceability meta-model

In their work they mention the tracing of requirements by means of a CASE tool, which uses an ADA Structure Graph to maintain the relationships [113]. The usage scenario is mainly change impact analysis. The authors provide a figure showing the relationships a typical requirement has with its environment i.e. the stakeholders, concepts, artefacts. With much empirical work, like surveys and focus groups Ramesh et al. try to build a meta-model for traceability. One of the results is the following figure, which nicely shows dependency between the main artefacts in requirements tracing.

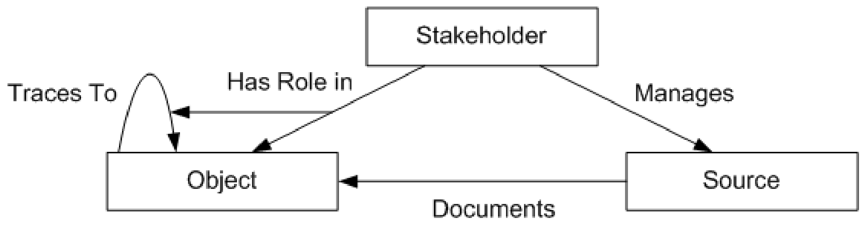


Figure 16 Traceability meta-model [112]

Trace Generation

Trace Generation deals with explicitly capturing relationships from requirements to other artefacts and dependencies between requirements. Trace generation [35], [59] is mentioned in almost all publications which I considered to be relevant for the topic. But also other terms have been used to describe this activity: synonyms for trace generation are trace capturing, and trace production which can be found in Von Knethen et al. [81] and in [63]. Their survey furthermore divides trace generation into the two activities (1) trace-capture, and (2) trace extraction (and representation). Trace capture directly maps to our activity trace generation. Trace extraction and representation means to provide stakeholders with a subset of trace information which meets their specific need. In [36] I find the activities "trace link identification" and "validation of trace link candidates" summarized under trace generation. Another term found is "link retrieval" by Cleland-Huang et al. in [21]. Also Hayes et al. use the term "link generation" [70]. Spanoudakis et al. refer to trace generation by the term "generation of traceability relations" [125].

Parameters influencing this activity are: information retrieval parameters such as recall and precision, coverage, completeness, number of queries, number of traces, number of contributions per author, number of iterations, Recovery Effort Index, average effort per unit to trace. Detailed information about the parameters can be found in section "Parameters for Requirements Tracing".

A good overview on the different trace generation approaches can be found in [21].

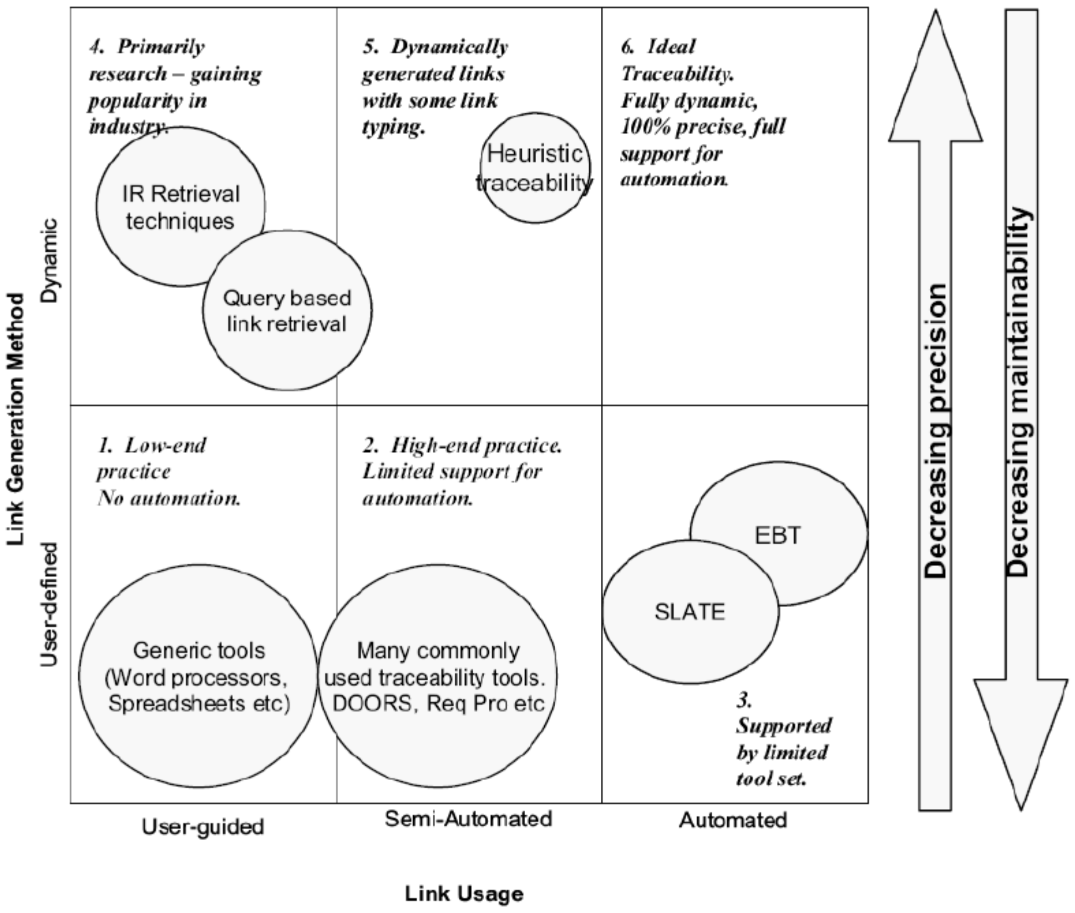

4. Primarily research – gaining popularity in industry.
IR Retrieval techniques
Query based link retrieval
5. Dynamically generated links with some link typing.
Heuristic traceability
6. Ideal Traceability. Fully dynamic, 100% precise, full support for automation.
1. Low-end practice No automation.
Generic tools (Word processors, Spreadsheets etc)
2. High-end practice. Limited support for automation.
Many commonly used traceability tools. DOORS, Req Pro etc
EBT
SLATE
3. Supported by limited tool set.
Link Generation Method
Dynamic
User-defined
User-guided
Semi-Automated
Automated
Link Usage
Decreasing precision
Decreasing maintainability

Figure 17 Current traceability practices

As already identified by Pinheiro [105] there are basically two ways to generate traceability information, I extend this classification with a third point, which leads us to the following list:

- Manual: refers to the continuous generation of traces during the development of the system as already postulated in [107]. Synonyms are “on-line tracing” [81] or “continuous tracing”.
- Automated: refers to the generation of traces after the relevant artefacts have been created [70]. Full automation is not possible at the moment, that’s why also with automated approaches some manual work is necessary. Synonyms are: “after-the-fact-tracing” or “offline tracing” [70].
- Heterogeneous: refers to the combination of the two approaches stated above [23]. Following I want to introduce the principles of three kinds of trace generation stated above.

Manual trace generation

Manual trace generation means that a team member explicitly decides that there exists a traceability relationship between two artefacts. There are several ways to manually maintain traceability information. These include the usage of COTS applications to edit lists and tables containing traceability links. Another possibility is to enter traces into a database. There it is possible to take the database of a widely accepted requirements management tool such as IBM RequisitePro or Telelogic DOORS or to make use of a standard database and customize it to the specific needs of the project team. The most obvious way is to store and manage traceability relationships in matrices. I call these requirements traceability matrices (RTM). Their advantage is that they can be generated and maintained with popular spreadsheet applications. Without much effort many of the stakeholders can start to read and manipulate traceability information. Requirements are written on the two axes of a two-dimensional matrix and dependencies are marked with a symbol (in our case an asterisk in the left part Figure 18). With this I am able to model multiple relationships.

	R1	R2	R3	R4
R1		*		
R2	*		*	
R3				*
R4			*	

Requirement	Depends-on
R1	R2
R2	R1,R3
R3	R4
R4	R3

Figure 18 Traceability Matrices (left) and Tables (right)

So we can see that a change in requirement R2 would affect R1 and R3. From Hayes et al we learn that CASE tools like DOORS and RequisitePro assist in building the requirements traceability matrix, but developers often do not build it to the proper level of detail [70]. Finally matrices are not just a way to maintain traceability information but also for representing automatically generated traces.

With a growing number of requirements (more than 250) matrices tend to be unmanageable [83]. For these cases traceability lists can be suitable, as depicted in the right part of Figure 18. Although they are on the first sight not as concise as the matrices, they offer better scaling if requirements get very numerous. Clearly also traceability lists do not scale to a very high number of requirements. However they are a good way to start tracing and do not require costly implementation of new tools and technologies.

Traceability databases

Another approach is to generate a requirements database entry and to add a linking attribute. Macfarlane and Reilly [90] also introduce the concept of link objects. The basic idea is, that the information about dependant artefacts is not stored on the artefact-side but in the link object itself.

Though I have mentioned requirements management tools (cp. Requisite Pro, Telelogic Doors) in connection with the RTM, their underlying technical concept is also a requirements and traceability database.

Also Romanski [117] reports the use of requirements databases and mentions direct links to the configuration management system. So the requirements database can evolve to a traceability database.

Graphical models

Certainly it is very convenient to have traces represented in a graphical environment as described by Von Knethen in [81]. But the introduction of such graphic models always requires committing to a specific tool. In our survey I could not identify a tool that fits in this category.

Tagging

Tagging refers to the practice of marking artefacts in a specific way, so that relationships can be derived from these tags. It is also a manual activity, but with the advantage that we can trace at a very fine level of detail (granularity) as we can assign these tags even at paragraph or line level [81]. Such a paragraph is subsequently preceded by tag of a special requirements tracing markup language, which identifies it as a subject for trace generation.

One approach for tagging is presented by Song et al. in [124] with their system STAR Track they are able to link specific sub elements of an artefact with other ones. The user has to mark the designated paragraph or code method with a tag out of STAR Track's specific syntax. Such a tag could look like the following:

```
<RE-1 Performance Requirement>
```

…where RE-1 is the id and Performance Requirement is the title. Tagged documents must be submitted to a web-based document repository / configuration management, where they are further parsed.

STAR-Track consists of three major components:

- Information Extractor: Extracts key software artefacts and their cross references from documents. Before the extractor can be applied to a document, the document must be tagged using the STAR-Track tagging method.
- Information Manager: Stores, searches, and retrieves information and references resulting from the extraction.
- Information Viewer: Provides a mechanism for querying and browsing the document information and references. By using a web-based solution for presenting the results, platform independence can be assured in this part as well.

Generally STAR-Track is similar to features provided by tools like RequisitePro but offers more freedom to the user in the definition of tag-ids.

Another application for tag-based tracing is the UNIX-tool RADIX proposed by Chi-Liang Ni [99]. It provides support for requirements enumeration and traceability. Typical Radix syntax would look like the following:

```
.dS R-40V2 doc01
This is the requirement R-40 text.
.dX
This is the R-40 explanatory text.
.dR 20 doc02
.dE
```

The printout is:

```
<R-docl-40V2>
This is the requirement R-40 text.
This is the R-40 explanatory text.
References: doc02-20
<End of R-docl-40>
```

On code level the same enumeration/tagging is applied. Here the tags for RADIX reside in the comment fields of the source code and are therefore not interpreted by the compiler, but by RADIX.

Furthermore a RADIX-plugin for Microsoft Word is available which eases the tagging of requirements documents with RADIX syntax.

Event based traceability (EBT)

In [20] and [19] Cleland-Huang et al. present an approach for implementing traceability links by using the event-notifier design pattern. Objects (such as code objects etc.) subscribe to their corresponding requirements and are notified about changes in their requirements. The subscription is a manual process, such as establishing traces in a matrix. However this should allow for the evolution of software projects.

The idea is that changes in the traceability relationships (e.g. the decomposition of a requirement in two smaller ones) can be reflected to the traceability database by means of event messages. The following figure illustrates the process.

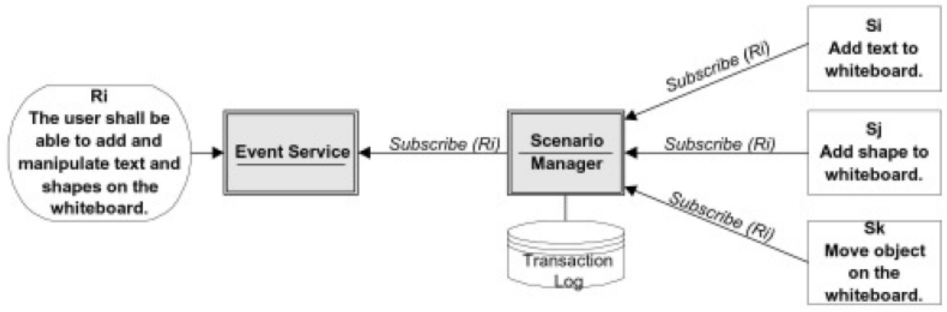
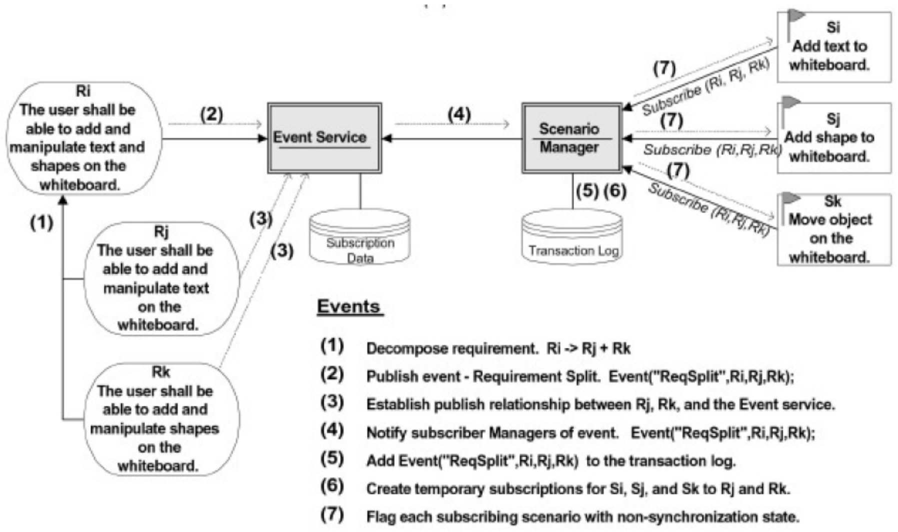


Figure 19 Initial configuration using an Event Service [19]

Changes in requirements are propagated like the subscription messages (e.g. Subscribe (Ri)) but in the other direction.


Ri
The user shall be able to add and manipulate text and shapes on the whiteboard.
(1)
Rj
The user shall be able to add and manipulate text on the whiteboard.
Rk
The user shall be able to add and manipulate shapes on the whiteboard.
(2)
(3)
(3)
Event Service
Subscription Data
(4)
Scenario Manager
(5) (6)
Transaction Log
(7)
Subscribe (Ri, Rj, Rk)
(7)
Subscribe (Ri,Rj,Rk)
(7)
Subscribe (Ri,Rj,Rk)
Si
Add text to whiteboard.
Sj
Add shape to whiteboard.
Sk
Move object on the whiteboard.

Events

(1) Decompose requirement. Ri -> Rj + Rk
(2) Publish event - Requirement Split. Event("ReqSplit",Ri,Rj,Rk);
(3) Establish publish relationship between Rj, Rk, and the Event service.
(4) Notify subscriber Managers of event. Event("ReqSplit",Ri,Rj,Rk);
(5) Add Event("ReqSplit",Ri,Rj,Rk) to the transaction log.
(6) Create temporary subscriptions for Si, Sj, and Sk to Rj and Rk.
(7) Flag each subscribing scenario with non-synchronization state.

Figure 20 A change event in EBT [19]

A change in requirements leads to a generic event message to the event server. The message contains structural and semantic information about the change itself, including the event type {Create | Inactivate | Modify | Merge | Refine | Decompose | Replace}, requirement IDs and descriptions, and additional links to rationale and stakeholder involvement [19].

With the possibility to transmit data values with these events, the traceability relationship offers more functions than simple dependency relationship. This concept was also postulated by Ramesh et al. in [112]. Normal traceability can be achieved just by examining the subscription status of scenarios or requirements; this is therefore not dependent to the EBT scheme.

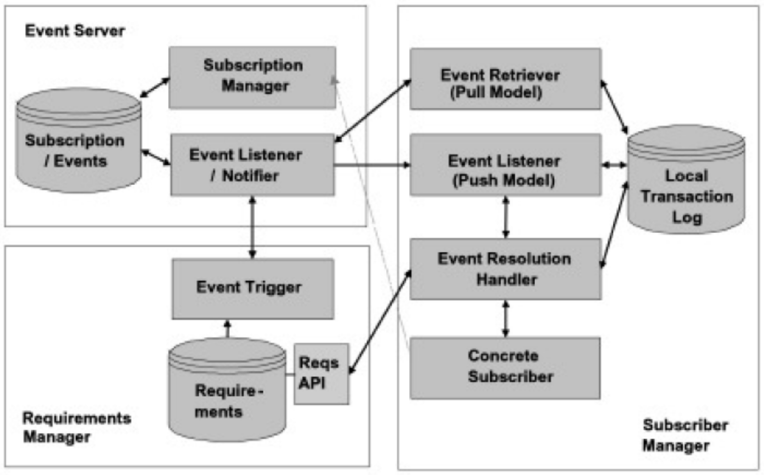


Figure 21 System model of Event-Based Traceability [19]

The prototype of event-based-traceability (EBT) was implemented on top of the DOORS requirements management system, to manually capture change events as they occur. The subscription interface was developed as a Java-servlet to support the subscription process over the internet.

The event server receives published events and processes them according to their subscription status in order to identify relevant artefacts (=subscribers) and sends the notification of change to them. The subscriber manager (a small server) then, receives event notifications on behalf of a particular type of dependent artefact. Events received from the event server are automatically resolved by the subscriber manager or stored in an event log for later semi-automated processing using a change processing tool [20][19].

Another implementation of EBT can be found in the work of Lang and Duggan in [85], who propose to apply it to a subset of requirements which is very volatile and critical (cp. value based requirements tracing [63]). They also interfaced to the DOORS-tool. Furthermore they mention the use of EBT for collaborative requirements engineering. Concerning maintenance they state that "EBT provides a more flexible and maintainable approach to long-term change through its use of the underlying publishsubscribe scheme"[85]. The authors state, that the EBT can also be used for larger systems as long as links are used strategically.

In principle, the publish-subscribe pattern is scalable, but the number of notification messages would become unmanageable if EBT was implemented at the highest level of detail for all design elements. However the authors motivate for strategic use of traceability.

From Lang and Duggan we finally learn that EBT led to enhanced communication between developers and end-users via the system, and also better end user understanding of the system [85].

Automated trace generation

In the last section I have shown valuable manual approaches, but they can become very labour-intensive. That's why researchers try to find ways to automate the process of generating traces. Characteristic for all these approaches is that the generation of traces is done after the artefacts have been generated, whereas manual tracing is mostly done during artefact creation.

Pohl gives a general statement for automation in [107] which is fortified by his small literature survey "Trace capture must be, as far as possible, automated. Manual recording would cause additional workload and manually recorded traces are often subjective and idealized." Therefore, traceability should come as a side effect of the developer's daily work. Main contributions in the field of automation come from [70][56][57][58][39][40][4][20][36][38][70] which are explained in detail below. This shows that research on requirements tracing is very much centred on this topic. Concerning the term I found "automated trace generation" as well as "after-the-fact-tracing" [70].

Hayes et al. break the problem down to a similarity analysis between requirements and design elements which fulfil these requirements [57]. The whole problem can be reduced to information retrieval [46], which deals with finding the all relevant links to a specific search query. It is somehow very similar to the use of an internet search engine [56].

The differences are that I am not searching for a homepage which contains our search term but for a requirement that contains words extracted from a specific source code element. Furthermore, I am always searching multiple terms in multiple datasets requirements, test cases, and source code.

So searching the requirements database (i.e. the requirements document) for the term "user account management" I might get three requirements. A search with the same term in the source code might return one class-file. Therefore I could establish a traceability relationship between the requirements and the single source file. There exist information retrieval algorithms to automate this activity. A prerequisite is that all artefacts to be traced are ready to be searched by a program i.e. they are machinereadable.

A database certainly enhances performance of queries which are necessary to build the traces. Following, I would like to introduce the concepts underlying automated trace generation. Certainly due to the complexity of this topic my introductions are just meant as a starting point. The interested reader might refer to the original publications for further details.

Table 4Main categories of automated tracing approaches

Method	Examples	Samples
Probabilistic Information Retrieval	The idea is to compute whether a document Di and a source code element Q have a traceability relationship. A measure for this is the similarity of these documents. The similarity of a document Di and a source code element can be modelled by the probability (Pr) that Di would be a relevant answer to a query for the words found in Q. Thus applying a probabilistic formula we get for each document Di a list of source code elements, ranked on the basis of the probability of being relevant for this document. So a typical result set for the document Di could be: Q1 has a probability of 0,95 of being relevant Q2 has a probability of 0,60 of being relevant Q3 has a probability of 0,55 of being relevant Q4 has a probability of 0,05 of being relevant A human "validator" then decides which of these candidates really has traceability relationship with the document. Most likely there will be a trace between Di and Q1 but Q2, Q3 and Q4 must be examined as well. An elaborate explanation of the underlying mathematical concepts of this approach can be found in [4].	[4], Poirot [89]
Vector Similarity	Starting from the document side, we could have a document that contains the sentence "The explosion destroys the vegetation". Looking at the frequencies of terms we would get a document vector like d = (0,..,2,…,1,…,1,…,1,…) "the" occurs two times, "explosion", "destroys", "vegetation" occur	Retro [70][56], Vanilla Vector [57]

	one time, and other word occur zero times) A query could look like "Destroys the explosion the vegetation?". Assigning the frequencies to our vector space (…,..,"the",…,"explosion",…,"destroys",…,"vegetation",…) this would lead to a query vector q = (0,…,2,…,1,…,1,…,1,…) which is identical with the document vector. Of course this is an idealistic example, in reality we are aiming not to find identical but similar vectors. Even from this little example we can see, that words which occur very often in a document are not so distinctive for its meaning (Cp "the" in our example above). They are mostly articles or conjunctions. Building a proper vocabulary is therefore an important task in when applying the concept of vector similarity. Mathematically we have the document vector d = (w1, w2, .., wn) and the query vector q = (q1, q2, .., qn). The similarity between these two is computed with the cosine angle between the vectors: $$sim(d,q) = \cos(d,q) = \frac{\sum_{i=1}^{N} w_i \cdot q_i}{\sqrt{\sum_{i=1}^{N} w_i^2 \cdot \sum_{i=1}^{N} q_i^2}}$$ Of course a significant amount of computation is necessary for this approach, as we have to calculate the similarity between every pair of artefacts.	
Latent semantic indexing (LSI)	LSI is basically an improvement of the vector similarity approach. It represents the documents by means of a matrix. Then singular value decomposition is applied to the matrix. The idea is to reduce its dimensions by generalising the meanings of words. In the ideal case synonyms are identified and thus reduce the complexity of the matrix. If we had for example the words dog, hound, and domicile, house it would be possible to reduce this vocabulary to the general concepts "dog" and "house". This should reduce complexity as well as computation effort for the similarity analysis. A prototypical implementation of LSI is done by Marucs and Maletic [92]. Their approach reprocesses the input file (source code, documentation items…) which means splitting the files based simply on well known coding standards. The authors consider their approach very promising and stress that their approach is language and programming language independent.	[70][56], Vanilla Vector [57]
Code Observation	A more semi-automatic approach for generating traceability links is to observe the code at run-time. For this we have to record which requirement we are currently observing, looking at the code parts which are triggered during the execution we can generate traces.	TraceAnalyzer [35], FastTlnC [49]

	Clearly we need the technical possibility to observe code at run time. Egyed's TraceAnalyzer [35] is such a system. He focuses on observing the execution scenarios instead of single requirements. I mention TraceAnalyzer here under the automated approaches, but in fact it needs some manual input (traceability relations) in advance to start the automatic generation.	

Basic measures for information retrieval

Below is a short explanation of the two most important IR parameters. I introduce them here, because I need them to explain the enhancement approaches for IR trace generation approaches. The respective formulas and further examples can be found in the parameters section.

- Recall: It is clear that with automated trace generation the primary goal is to fully cover all links which one would also manually create. This is measured by a measure called "recall" typical recall values are around 90% [21].
- Precision: Unfortunately automated approaches also generate links where there should not be any. This is measured by precision which indicates how many true links are among the automatically generated set. Typical precision values are around 20% [21]. From a small literature survey in another work I get more scattered values with precision rates typically at 10-60% when the threshold level is set to achieve 90-100% recall [22].

Especially because of the comparably low precision values automated approaches still need a considerable manual rework. A goal for this field is therefore to take the burden away from the people which are required to manually check the links, and therefore increase precision while not harming the good recall values.

Improvements for automated trace generation

Very basic information retrieval algorithms are still outperformed (in terms of quality) by human analysts generating traceability links, there is just too much rework involved. That is why there exist some approaches to improve such algorithms by combining them with keyword based methods.

- The first improvement, retrieval with key phrases, is to associate a list of technical terms with the document repository. As an example a traditional keyword algorithm would generate a link between the terms "abc metadata exchange" and "abc test environment" because of the term "abc" But by adding the phrases above to the key phrases we might be able to decrease the importance of this (potentially irrelevant) link. On the first sight this seems very much effort, but often such a list of terms can be found in the appendix of a requirements document [57]. A similar improvement is mentioned in with the term graph pruning: Sometimes the same term is used to describe very different concepts. If our signal term was "schedule" it could describe could describe the maintenance interval of a truck as well as the backup interval for data. The idea of graph pruning is to prevent the creation of links problematic areas at the beginning. Pruning utilizes initial decisions made by the analyst against the training set data, in order to discover where to place constraints and to improve the precision of problematic areas. Rather than attempting to establish individual constraints which would take excessive effort, the approach of Huang et al. [21] creates constraints between groups of requirements and groups of documents [21]. This is an activity may also be conducted in the Specification Phase. Graph pruning seems to improve precision dramatically but the problem in first studies was that recall dropped below 90% which was considered unacceptable [21].
- The second improvement is a thesaurus based approach. It is an extension of the keyword

list idea. Starting from two terms e.g. "corrupted data" and "missing packages" a similarity coefficient is assigned to them in this case we could take 1.0 as these terms are connected., so this breaks down to the task of assigning the coefficient to the pairs [57].

- Relevance filtering: From our automated trace generation approach we get link candidates with relevance values attached. The weakest 10% of the traces contain 90% of the false positives. Thus, another improvement is to omit links with relevance values below a certain threshold [56]. This greatly increases precision but at the price of recall. A similar concept is the "strength thresholds" introduced by Egyed et al in [36].
- Hierachical enhancements: This approach takes in account that requirements are always hierarchically organised. This could for example be the section titles in the requirements document. This also implies some requirements inter-dependency. In [21] I therefore find an approach to modify the retrieval algorithm to make use of this structural information
- Logical clustering enhancement: This is another improvement for automated trace generation. The basis is the assumption "that if a link exists between a query and a document and if that document is part of a logical cluster of documents, then there would be a higher probability that additional links should exist between the same query and other documents in that cluster." [21] The results of this enhancement were that precision could be improved by 5%- 10% depending on which specific way of clustering was applied.
- Graph pruning enhancement: Automated approaches tend to place references because of the occurrence of a specific word. But this also leads to many false positives.

Especially the thesaurus approach seems to be very promising. Hayes et al. compared these different approaches against each other: Evaluation of the algorithms against a comparable keyword-based tool and analysts showed that the retrieval with thesaurus algorithm outperforms all in terms of recall and sometimes - in terms of precision [57].

RETRO

Huffman-Hayes et al. present RETRO (Requirements Tracing On-target) which is a special-purpose tool, designed exclusively for requirements tracing. It can be used as a standalone tool to discover traceability matrices. It can also be used in conjunction with other project management software: The requirements tracing information is exported in a simple, easy-to-parse XML form. Underlying is an information retrieval (IR) toolbox, implemented in C++; several IR methods have been adapted for the purposes of the requirements tracing task. These methods are accessed from a Java GUI block to parse and analyze incoming requirements documents and construct relevance judgments. A Filtering/Analysis component (implemented in C++ as well) processes the list of candidate links with one of the methods and provides the analyst with a list of them [70][56]. Retro fulfils the following tasks:

- Extraction of high-level (requirements) and low-level (code) elements form the source document and converting them to a format readably by RETRO
- Parsing and stemming of high and low-level elements by a special IR algorithm
- Removing of common stopwords (words to exclude) such as "and", "a", "the" etc.

Finally I find the statement that high levels of precision are only possible with the help of filtering.

Poirot

An example system for automated trace generation is Poirot [89] a tool which makes use of an underlying probabilistic algorithm to generate traces. Poirot is typically able to obtain 15-30% precision while achieving recall rates of 90-95% [21]. Poirot uses a server based approach: In the Poirot server reside Trace Manager, Trace Engine, Data Access, Controller, Resource Manager, and Resource Broker. The server can be connected to CASE-Tools by means of an XML-interface. So one could make use of the date stored within Requisite Pro or DOORS. The authors also state that visual data (e.g. sequence diagrams) can be processed by means of an SVG (scalable vector graphics)-adaptor.

TraceAnalyzer

Egyed's scenario-driven approach [39] [35] also distinguishes several phases for trace generation and validation. The scenario-driven approach is a matching between development models and the actual implementation. The TraceAnalyzer is a parsing and graphing tool which makes use of a commercial software monitoring tool in this case Rational PureCoverage. PureCoverage is needed to observe the system during the execution of specific scenarios. By this means traces can be established between the scenarios and the system. The author refers to the sourcecode as the footprint of the system.

Dependencies between the software artefacts themselves are discovered by this means, for this TraceAnalyzer examines whether the code of two artefacts overlap during execution in some way. A prerequisite is that an executable version of the software is already available. And a constraint in this approach is that it cannot relate dependencies among model elements that do not relate to code. With COTS-tools for monitoring software systems the system is observed during its execution. TraceAnalyzer tries to deal with this challenge of non-functional requirements with the assumption, that non functional requirements can also be traced to several implementation classes. Therefore TraceAnalyzer offers a means to link a user need to multiple requirements and therefore also show the affected implementation classes [40].

In summary, TraceAnalyzer is a tool which automatically generates traces on the basis of scenarios but needs some initial manual input as a baseline. The iterative nature of the process enables continuous improvement of coverage and trace quality. Furthermore Egyed admits that testing is a validation form that cannot guarantee completeness, as there may be some missing test cases. For this TraceAnalyzer offers an input language, where the user can express uncertainties [39]. An application of the TraceAnalyzer technique can be found in [40], where this approach was tested on a video-on-demand system.

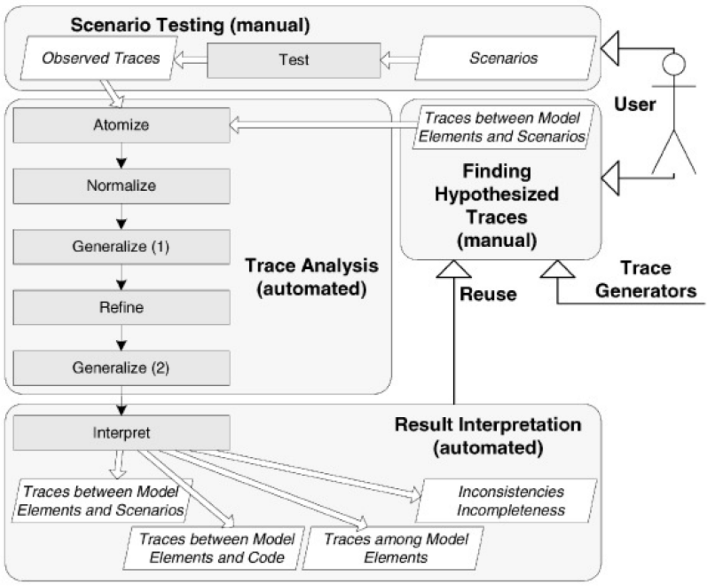

Scenario Testing (manual)
Observed Traces
Test
Scenarios
User
Atomize
Normalize
Generalize (1)
Refine
Generalize (2)
Trace Analysis (automated)
Traces between Model Elements and Scenarios
Finding Hypothesized Traces (manual)
Reuse
Trace Generators
Interpret
Result Interpretation (automated)
Traces between Model Elements and Scenarios
Inconsistencies Incompleteness
Traces between Model Elements and Code
Traces among Model Elements

Table 5 Scenario driven approach for traceability [39]

1. Hypothesising: reasoning about the traces that could exist by examining static documentation

2. Atomising: observation which scenarios make use of which classes and methods of the system. Out of this a so-called footprint graph is generated. This graph is hierarchically structured.
3. Generalizing: hierarchical decomposition of the footprint graph which means starting to derive further trace information from the "leaves" of the graph -> further traces are discovered
4. Refining: is the reverse activity to generalizing, here the "roots" of the graph are examined. Changes and discoveries are then propagated downwards in the graph structure.

Trace generation to design models

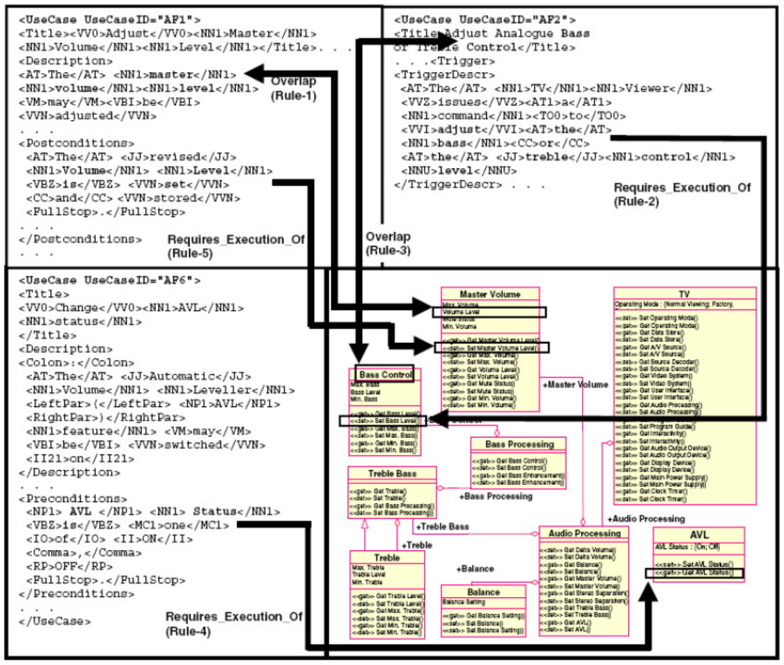
Spanoudakis et al. present an approach for generating traceability relations between a requirements document (or use case documents) and design models (analysis object models) [125]. The requirements document defines functional and non-functional requirements in natural language and in broad terms. Use cases are composed in natural language as well, but they are more structured and define in detail different ways in which the user can use the system.

Design models such as UML in comparison are already a technical view on the future system and give very detailed information about the implementation of the system.

The approach of Spanoudakis et al. now aims at creating traceability relations between use cases and the design model with the following process:

1. Grammatical tagging of the textual requirement statement and use case documents: the documents are processed by a general-purpose hybrid grammatical tagger. This tagger identifies grammatical roles of the words in the text of these documents. The goal is to resolve ambiguities.
2. Conversion of the tagged requirement statement and use case documents, and the analysis object model into XML representations: a. The requirement statement and the use case documents are tagged according to a special document type definition the XML representations are structured by two different groups of elements. The first group is used to structure the documents such as the <preconditions>-tag. The second group is used to render more precisely the specific meanings of the single words. E.g. <NN1> for a noun or <VVI> for a verb. b. The object analysis model (UML) is converted into XMI format by using the Unisys XMI exporter for Rational Rose.
3. Generation of traceability relations between the requirement statement and use case documents and the analysis object model: the content of all XML-documents is analyzed by means of a rule engine. With additional rules additional constraints may be specified. Also the usage of a synonyms lexicon is mentioned. These relations are then also stored in XML similar to XLink.
4. Generation of traceability relations between different parts of the requirement statement and use case documents. For this the content of the documents is not analyzed. They are just linked by so-called IREQ rules. These rules generate relations between these documents only if they are connected with particular combinations of other traceability relations with the same elements of an analysis object model. So this is the concept of deriving relationships between two requirements from the fact that they trace to the same artifact.

The result of such this process could graphically look like the following:



```
<UseCase UseCaseID="AF1">
<Title><VV0>Adjust</VV0><NN1>Master</NN1>
<NN1>Volume</NN1><NN1>Level</NN1></Title>. . .
<Description>
<AT>The</AT> <NN1>master</NN1>
<NN1>volume</NN1><NN1>level</NN1>
<VM>may</VM><VBI>be</VBI>
<VVN>adjusted</VVN>
. . .
<Postconditions>
 <AT>The</AT> <JJ>revised</JJ>
 <NN1>Volume</NN1> <NN1>Level</NN1>
 <VBZ>is</VBZ> <VVN>set</VVN>
 <CC>and</CC> <VVN>stored</VVN>
 <FullStop>.</FullStop>
. . .
</Postconditions>
. . .
```

```
<UseCase UseCaseID="AF2">
<Title>Adjust Analogue Bass
or Treble Control</Title>
. . .<Trigger>
<TriggerDescr>
 <AT>The</AT> <NN1>TV</NN1><NN1>Viewer</NN1>
 <VVZ>issues</VVZ><AT1>a</AT1>
 <NN1>command</NN1><TO0>to</TO0>
 <VVI>adjust</VVI><AT>the</AT>
 <NN1>bass</NN1><CC>or</CC>
 <AT>the</AT> <JJ>treble</JJ><NN1>control</NN1>
 <NNU>level</NNU>
</TriggerDescr> . . .
```

```
<UseCase UseCaseID="AF6">
<Title>
<VV0>Change</VV0><NN1>AVL</NN1>
<NN1>status</NN1>
</Title>
<Description>
<Colon>:</Colon>
 <AT>The</AT> <JJ>Automatic</JJ>
 <NN1>Volume</NN1> <NN1>Leveller</NN1>
 <LeftPar>(</LeftPar> <NP1>AVL</NP1>
 <RightPar>)</RightPar>
 <NN1>feature</NN1> <VM>may</VM>
 <VBI>be</VBI> <VVN>switched</VVN>
 <II21>on</II21>
</Description>
. . .
<Preconditions>
 <NP1> AVL </NP1> <NN1> Status</NN1>
 <VBZ>is</VBZ> <MC1>one</MC1>
 <IO>of</IO> <II>ON</II>
 <Comma>,</Comma>
 <RP>OFF</RP>
 <FullStop>.</FullStop>
</Preconditions>
. . .
</UseCase>
```

Figure 22 Traces from use cases to UML diagrams [125]

The authors also mention that classic keyword based method such as those presented by Antoniol et al. [4] could also be used for their tasks. But they criticize that these methods are not capable of identifying the semantic meanings between the textual documents. Furthermore they mention the relatively low precision which comes with IR.

Exposure of tacit knowledge

Concerning pre-requirements specification traceability Stone et al. present an approach to generate implicit knowledge which is not included in the specification yet [127]. The idea is that many ideas from elicitation activities do not get into the specification. So their Latent Semantic Analysis (LSA) approach splits the specification into small chunks and the source material from the elicitation activities too. Then it tries to build traces between them and shows for which chunks of the source material traceability links do not exists. This leads to the exposure of tacit knowledge.

Data Warehouse approach

Another interesting approach is reported by Neumüller et al. [97]. In their case study the report that in the company which was required to introduce traceability, the majority of information was already residing in an Oracle database (such as documentation wiki, data model, and server-side PL/SQL code) so the main task was to develop import agents for other systems. The main import tasks are conducted by an awk script scheduled as cronjob.

“The tracking system already allowed to hierarchically organize requirements and to manually define links. The majority of the trace links was generated by utilizing existing company naming conventions and customs. Developers were already using the id from the tracking system in source code comments and CVS logs. By isolating the ids and attributing them to the appropriate subprocedures of PL/SQL program modules, we were able to define relationships between these artifacts.” “the application relies on conventions and uses simple name lookup to provide links between these artifacts” “Ensuring an acceptable quality of links without constricting the developers too much, however, still remains a challenging research issue.”

With the citations above it is clear that such a system could only be successful where elaborate naming conventions (also in the specification/requirements document) are in use. This is something that must be defined in the specification phase.

Neumüller et al. give some useful hints what should be kept in mind when introducing requirements traceability in an organization:

- Introduce traceability features incrementally: no big-bang approach, convince engineers subsequently
- Provide smooth integration with existing tools: avoid "yet-another –tool", on top and tightly integrated with the existing tools.
- Automate the right set of features: do not adapt every feature of COTS tool, but automate those with the highest value
- Make only small changes to work practices of developers: tailor traceability strategy to the specific needs of development.
- Focus on already established development practices: do not introduce too much at one time.
- Emphasize trace utilization: provide query mechanisms and visualization of traces, so that the information is really used.

Generally the work of Neumüller et al. shows how the introduction of traceability can work for small companies as well. The link-generation on the basis of existing naming conventions is self-evident, but easy to implement if such conventions are already present. They mandate a custom build system to develop a traceability infrastructure which is successful from a value-based perspective and creates the biggest value for the engineer. Small companies are more flexible and this is the advantage they have to leverage.

Automated approaches are a very interesting way to generate traces. No extra burden for the developer is introduced and traceability is just a side effect. Nevertheless the term "automated" might be somehow misleading, as there is manual rework involved in every approach. A main goal for the future is to further prevent the generation of false positives.

Heterogeneous tracing approaches

As we have seen there are many different approaches for traceability. They all have their strengths and weaknesses, so it suggests itself to combine these to a best of breed solution. This is what can be subsumed under the term heterogeneous traceability. Such an approach is proposed by Cleland-Huang et al. in [23] by the name of Traceability for Complex Systems (TraCS). See figure 13 for the structure of TraCS. We can learn from the figure that TraCS combines very "traditional" traceability practices, such as an RTM, with automatic ones such as link generation by information retrieval and event based traceability. To make the mix of different trace approaches more tangible, I would like to show the categories, used in [23].

- Substantial: Trace 50% of the most critical requirements explicitly. Trace NFR goals related to performance, security, and safety etc using Event Based Traceability (EBT) (< 5%). Trace remaining requirements using dynamic IR methods (45%).
- Significant: Trace 5% of the most critical requirements explicitly. Trace < 2% of NFR goals using EBT. Trace remainder of requirements using IR methods.
- Nominal: Trace all requirements using IR methods.

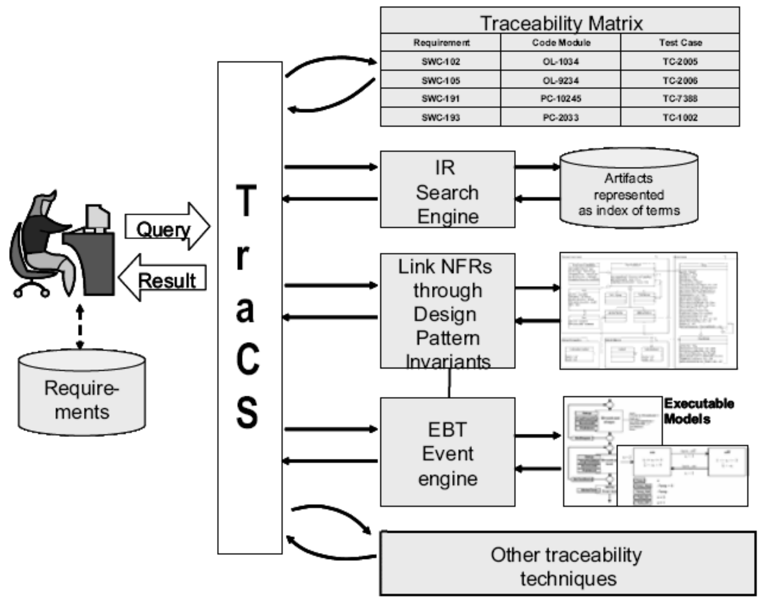


Figure 23 TraCS

Many approaches are necessary because they were assigned to the risk categories, which were also named Substantial, Significant and Nominal. For this the measure “Failure to trace” (explained in the parameter chapter), was used. The use-cases are then partitioned in these categories so one can decide which trace strategy to use for a specific use case.

The advocate of heterogeneous traceability state, that there is still further empirical work necessary to determine whether it also works in an environment beyond prototypical systems. But generally the approach seems very promising. It is clear that this approach involves more work on the beginning of the project. Partitioning of requirements by means of criticality and defining of a trace strategy are part this additional effort (especially the trace strategy is a very reasonable idea). Furthermore one obviously needs some tool to conduct the different traceability approaches. While it might be easy to introduce tracing by means of a traceability matrix, creating an information retrieval system for automatic traceability could be a little more difficult. So this is perhaps an approach for organisations where different tracing approaches are already in use.

Trace Deterioration

After the generation of traces it is likely that the traceability information will get out-dated over time. Changes in requirements in the design or in the implementation itself might therefore lead to a deterioration of traces. Thus trace deterioration is not an activity, but continuous little events that affect the timeliness of traces in a negative way.

Deterioration is often referred to under the term evolution of traces [111][105]. Requirements evolution is a critical aspect of software development. Therefore appropriate support for the evolution seems to be crucial for the success of any traceability system.

Also Cleland-Huang et al. point out that “traditional traceability methods have a tendency to deteriorate, as time-pressured practitioners fail to systematically update impacted artefacts and links in a timely fashion” [19].

We could also see the deterioration of traces as aspect of the evolution of requirements. Ramesh et al. [112] note that all of their focus group participants stated that a traceability scheme should help document and understand the evolution of requirements [112]. While the introduction of new requirements is a different issue, the purging of out-dated ones is a definite application of an activity (rework) that has to be enacted when the deterioration event took place.

I have identified some approaches that explicitly deal with the problem of out-dated traces and will therefore mention the right below.

ArchTrace

This is a tool which especially focuses on the evolution of traces that is why I present it in this section. Generally many tools focus on a snapshot generation of the traceability relationships in a software project. Murta et al. developed a tool called ArchTrace which aim to capture traceability information over several versions of the system [95]. Keeping in mind the evolutionary nature of software this seems to be a very reasonable approach. ArchTrace must be provided with an initial set of valid traceability links. This seems similar to the TraceAnalyzer [35] approach, but ArchTrace needs full coverage of all present links and not just a starting-data. The developers state that this initial trace-set should be generated by means of data mining, information retrieval or syntactical analysis. With this basic information ArchTrace is connected to the configuration management system of the project. Relying on several policies the system now tries to reflect changes in the code or architecture to the traceability database. The following figure should provide a tangible example of this:

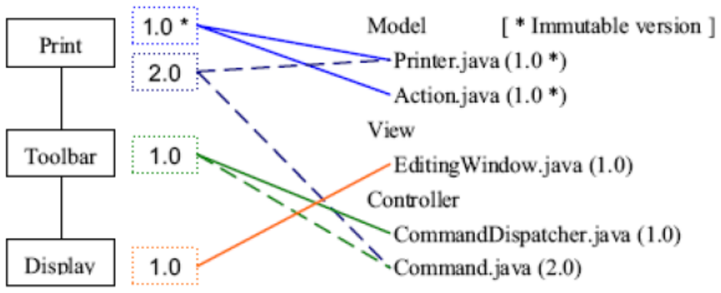


Figure 24 Trace changes by evolution [95]

We see, that the "Print" Requirements traced to Printer.java and Action.java in version 1.0. In version 2.0 the trace to Action.java is eliminated and a new trace to Command.java is introduced. This is an application of "Trace Deterioration", though traces do not only deteriorate, but new ones are introduced as well. The architecture of the system is as following:

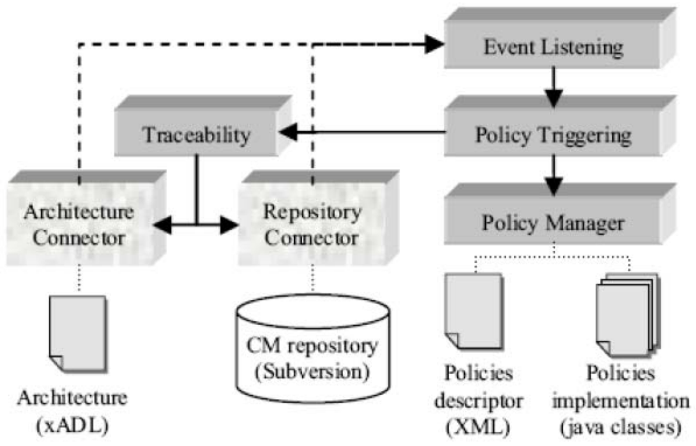


Figure 25 ArchTrace Architecture [95]

Especially the connection with the subversion CM seems to be very elegant, as it also enables collaborative working. Grey boxes are standard components and patterned boxes are custom components. The architecture is described with xADL [27].The authors report achieving a recall of 89% and precision of 95% in a case study of the system where the development of the Odyssey [134] system was historically replayed for two years. Though it seems impressive, we have to admit, that the success relies mainly on the quality of the initial data set of traceability information. In this case it has been manually generated by the developers. Generally the composition of existing components such as CM, Architecture Description Language, policies and event infrastructure seems very promising.

Trace Validation

Knowing that traces deteriorate it is clear that I need a downstream activity to determine which ones are still relevant and which ones should be up dated. As I have seen trace validation is necessary in nearly every tracing scenario as even automated approaches need some analyst feedback on the validity of the generated traces.

Parameters relevant for this activity are: precision, recall, strength of trace link candidate, relevance (probability) of links. Detailed information about the parameters can be found in the appendix.

From Hayes et al. we learn that validation is essential also in fully automated approaches After the automatic generation of traces a human still has to examine the traces for correctness [57][56][70]. Onlyby improving the candidate link lists, we can reduce the burden on the analyst. The incorporations of user feedback in automated trace-generation are therefore crucial for their success [70]. Egyed's scenario driven approach includes an explicit validation phase as well [39].

For the validation step after generating trace link candidates it is helpful to identify the candidates that are most likely to be false positives [36]. The basis for this decision is the measure "strength of trace link candidate". Again I encounter the concept of filtering [70][36]. (Cp. Improvements of automated trace generation)

But why using automated approaches if one still has to manually evaluate the links created by them?

An example can be found in [22]. There Cleland-Huang report, that an analyst had to examine 22 candidate-links between two classes. But without automated trace generation via information retrieval, the same analyst would have been forced to consider 360 possible trace relationships between the two classes.

In literature validation is also found for assessing how well a specific approach works. For example Antoniol et al. report, to develop a traceability matrix for being able to control precision and recall metrics [4]. So when developing a new automated approach it will be mandatory to manually validate its performance even if a validation phase is not planned for the final usage scenario of the approach. For such a validation it is therefore recommendable to acquire existing data sets where a full requirements traceability matrix (RTM) is available to validate the approach.

Trace Rework

Having discovered potential errors during the validation phase, it is clear that we also want to correct them in some way. For this I have the "Trace Rework"-activity. It is often performed together with the "Trace Validation"-activity. In Von Knethen's survey [81] they refer to this activity by the term trace maintenance. Their finding was that it is hard to manually maintain traces in spreadsheets, if extensive hierarchies are managed. Tool support is therefore essentially for this activity as well. Generally, rework is often not explicitly mentioned, but a sub-aspect of the validation process. A possible consequence for the TAF could be to merge the activities "Validation" and "Rework" to one single activity.

Ramesh and Jarke [112] mention several ways in which a requirement or trace can be changed.

Especially, the distinction whether the state of the requirement before the change is kept or not is interesting. To keep the original form of the requirement would be rather a "Replace" action. The same phenomenon happens when a requirement is deleted. One approach is to delete the requirement and to completely lose track of it. The other approach is to mark it as "Abandoned". It seems reasonable to keep a reworkhistory, so an "undo" of false changes is possible.

Event-Based Traceability (EBT)

Also the rather sophisticated approach of event-based traceability requires manual rework in sometimes. In this case event messages are stored in an event log to await manual intervention [20].

Rework influences maturity

An influence of trace rework is mentioned by Akley et al. in [9]. By calculating the metric Requirements Maturity Index RMI the experienced sudden declines of requirements maturity when conducting a major rework session on their requirements after four months. The same was true for a minor rework seven months after the project start. So I can say that rework temporarily harms the RMI but after the rework the level usually returns to its initial value again.

Trace Usage

Finally, I also want to leverage the efforts of the whole tracing effort. In the usage phase traceability information is taken as an input for decisions or other purposes. It is interesting to see that in many publications trace usage is not seen as part of a process. Although Von Knethen's survey [81] defines process phases such as trace definition, trace production and trace maintenance it is interesting to see, that they to not explicitly mention a usage phase. Often usage scenarios for traceability information are just mentioned in the introduction to motivate for the topic. Interesting is also that papers identify very different applications for traceability.

It is essential for the use of traceability information that it is tailored to the needs of its user. While a requirements traceability matrix (RTM) might be of no use for the management, some simple key figures (e.g. percentage of requirements covered by test cases) could provide them with useful information. So the selection of the right data and the appropriate means of presenting it is the key success factor in the usage phase.

Parameters relevant for this activity are: benefit, return on investment, risk reduction, quality improvement. Detailed information about the parameters can be found in the appendix.

Following, I want to list trace usage scenarios in a complete and adjusted manner to give a good overview on the numerous purposes of RT. Afterwards I will shortly describe each scenario.

Table 6 Applications of Traceability Information

Usage Category	Goal
Change Management	• Change impact analysis, to determine how many artifacts would be affected by a potential negotiated change. [56][55] [9] [40] [31] • Performance impact analysis, to determine the impact of a change on the performance of the whole system. [20] • Risk assessment, to see whether critical parts of the system are affected by a change [56]
Documentation Reengineering	• Justification for design decisions, by maintaining traces back to the rationale which led to the respective design. [9] • Reuse facilitation, by recording traceability information. [9][4] • determine what an artifact was initially supposed to do.

	[137][39][33]
Verification & Validation	• Coverage of test cases [9] [40] [56] • Coverage of requirements specification [112] • Avoid gold plating [133] • Track project progress [112][17] • Design satisfaction • Consistency checking
Requirements resolving	• traces between requirements: to detect potential conflicts [38] [40] • forward traces: to detect whether the underlying requirements are conflicting or cooperating [38] [40]
Collaboration	• Maintain traces to authors or adaptors of an artifact for concurrent collaboration or later adoptions. [124][3]
Program Comprehension	• requirements tracing: Provide all stakeholders with means to browse both in forward and backward directions [113] • program understanding: [86]

Tracing for change management

Cleland Huang et al. [20] distinguish two kinds of change:

- Contextual change: the setting of the system is altered. E.g. the number of users doubles.
- Functional change: the behaviour of the system is changed in a significant way. E.g. introduction of a new set of features

A classic use case is change impact analysis Here trace information is used to determine how a change to one artefact affects the others and support decision making [22]. Typically a requirement changes which makes change on a software component necessary. With traceability information we can now see how the change affects the other software components and also estimate the cost of change. The following things can be estimated:

- Cost estimation: As mentioned by Watkins and McNeal [133]. Here we aim to tell how the implementation of a specific change to the system would affect the project budget.
- Performance estimation: As mentioned by [20]. This offers a more technical perspective to change. Here event-based traceability is applied to see how a change would affect the overall performance of the system. Especially EBT deals with inserting speculative data values in performance models, to see how these models would perform under the proposed change.
- Risk assessment: to see whether critical parts of the system are affected by a potential change [56]

On one hand we need the forward-from requirements information on the other hand information about requirements inter-dependency is necessary. So the simplest way to of change impact analysis would be just to highlight the artefacts which would be affected by a potential change. A more sophisticated way could be to quantify the change necessary or to add complexity values to the respective artefacts.

Haumer et al. [55] mention that besides the traceability problem also an "envisionment problem" exists which makes changes difficult. The envisionment problem means that "people have great difficulties to envision the impact of a proposed system in a future that is vastly different from the present" [55]. They postulate that solutions for requirements traceability and envisionment have to be combined to cope with continuous requirements change.

Tracing for documentation

Trace information is being generated primarily for documentation. A possible scenario is the development of software components. During the project no trace information is used, but a policy or customer requires recording traceability information. This information can be useful if the component has to be adapted years later and the development team of the component has been dissolved.

Arkley et al. report in [9] that traceability is also used as a means to record the justification for design decisions. This involves tracing back to the rationale of a requirement (backward-from). Often requirements from existing systems or products are reused to build new products. Problems can arise if no information is available whether the reused requirements have any dependency to other requirements which are not reused. So requirements inter-dependency is a major issue in this case. I found an example for this in Arkley et al. [9] who refer to the benefits of requirements traceability for component reuse: traceability systems enabled the engineers to determine which development components (requirements, design elements and tests) could be reused, by comparing the cus omer's specification with previous specifications and identifying related design elements (or other project artefacts).

The reuse scenario is also identified by Antoniol et al. as they state the need for "...means to trace code to free text documents are a sensible help to locate reuse-candidate components" [4].

Tracing for requirements resolving

Another interesting application for traces is mentioned by Egyed et al. in [38] and [40]. They use traceability to identify conflicts between requirements. While it is obvious to use direct links between requirements to see whether they are conflicting or not, the new idea is to use forward to traceability links for the same purpose. The basic idea is that if two requirements trace to the same artefact (e.g. code) they are dependent to each other and might be either conflicting or cooperating. The approach is connected with existing work on the influences several non-functional requirements have on each other.

Having classified the requirements in categories like "Efficiency" "Usability" it is even easier to see which of the requirements are connected. The traceability approach stated above supports this process.

Tracing for verification & validation

Haumer et al. report that traceability was initially mandated by procurement agencies for purpose of compliance verification [117][55]. Thus tracing is used to verify whether all requirements in the requirements document have their counterpart on the software side. Furthermore one can also go in the opposite direction to see whether only things specified have been developed. (Avoid gold-plating) [133]. With a complete list of validated requirements it is even easier to estimate size and scope of the entire project [67]. Chisan et al. [17] point out that "Traceability established within the requirement specification which were used by project tracking to monitor resources, helped to prevent creep from significantly affecting progress."

Ramesh et al. [67] note that the traceability matrix also provides an extra value with the possibility to track staff process and completion status. Hayes et al. refer to this application as "completeness analysis"

Have all high-level requirements been fully satisfied? [56].

Variants [56]

- Verify design satisfies requirements
- Verify code satisfies design
- Validation that requirements have been implemented and satisfied

- system level test coverage analysis [56]
- regression test selection [56]

Tracing can also be applied to the testing side [9] [40]. So we generate traces to the test cases and can therefore determine how well the status of development is covered by test cases. This aids project managers in handling the cost and the customer confidence in the product is enhanced [133]. Arkley et al. report this as: The most important progress metric provided by the traceability system was determining when the development of the software was complete. This was achieved by demonstrating that all the requirements had been tested, by following the trace relationships from the requirements to tests, test cases and finally to each test step and the associated validated test result [9].

The usage of trace information depends on the type of stakeholder and the phase of the software development process, i.e. traces are not used as they were recorded and therefore selective retrieval according to the actual needs must be supported. Second due to the large amount of information produced during process performance, only content oriented trace capture provides the basis for appropriate reuse [18], i.e. the reuse of the recorded information is almost impossible if the trace objects are not embedded in a broader context.

Third, the people involved in the trace capture are most often different from the users of the trace information [107]. Therefore for efficient trace usage it is essential that every user group can work with a subset of trace information which is relevant for them. So some kind of "views" on the traceability information would be beneficial.

Tracing for collaboration

Tracing is primarily used to trace to the author of an artifact. Here trace information can be used differently. From a telephone call or an email to discuss changes with the responsible person for a respective trace thru an intelligent notification system to coordinate this communication different approaches are possible. So in distributed software development this can simplify collaboration, as there cannot be many meetings to discuss responsibilities. Also Lang and Duggan [85] developed their "RMtool" with a focus on supporting the collaboration between stakeholders in the requirements engineering process. They mention limited collaboration capabilities as a shortcoming of CASE tools which were available in 2001. Trace for collaboration is also mentioned by Ramesh [110], who also points out the use of traceability information in the domain of knowledge management by linking various sources of information. Especially the recording of tacit knowledge is mentioned.

Ramesh et al. refer to this trace application as accountability in [111]. They warn that "The use of accountability information as a means for performance appraisal may be inappropriate." And "Traceability needs to be something that humans can work with not just a whip held over people." "Accountability needs to be supplemented with good communication."

Tracing for reengineering

Starting form a legacy system with poor documentation, tracing is done in the reverse direction. From existing source code trace are generated to a requirements document. Or a new requirements document is reengineered from the source code. I find this usage schema in the work of Yu et al. [137] who report to identify non functional requirements with the traceability between code and goal model. Also Egyed states that his scenario-based approach may be used for reverse engineering of legacy systems [39].

Antoniol et al. refer to this application schema simply by the term "Maintenance" and "Design recovery" respectively [4].

Ebner and Kaindl [33] also report trace generation form a reengineering project. Main findings of their case study were:

- 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 the long run, which would include maintenance.

They achieved trace generation with the Requirements Engineering Through Hypertext (RETH)[76].

Tracing for program comprehension

A more informal application is mentioned by Antoniol et al. who point out that traceability eases the programmer's understanding of the system, especially when being able to trace back from the code back to its, specification or even rationale:

"Traceability links between areas of code and related sections of free text documents, such as an application domain handbook, a specification document, a set of design documents, or manual pages, aid both top-down and bottom-up comprehension" [4].

Lange and Nakamura's work is also in the field of tracing for program understanding [86]. They mention the value of traces especially in the field of object oriented design, which sometimes makes programs harder too understands. Here program visualization can aid programmers in understanding the execution of a program.

Pinheiro's [106] system TOOR relies of the formal specification language FOOPS (Functional and Object Oriented Programming System). TOOR is meant to be used during development than a-posteriori. Concerning usage TOOR provides different tracing modes:

- Selective tracing: restricts tracing to certain selected patterns of objects and relations
- Interactive tracing allows interactive browsing through the objects related to a selected object. The choice of forward and backward tracing can be changed at any time.
- Non-guided tracing allows a user to go from on object to another at will, inspecting contents as desired.

TAF Feedback cycles

One feedback cycle for the domain of automated trace generation is mentioned in [57]. The authors think that it would be beneficial if the author had the possibility to influence further generation of traceability links, with his opinion.

Another mentioning of feedback cycles can be found in [39]. Egyed states the belief that trace information should be generated iteratively contrary to the old approach develop first document later. In his approach previously detected traces become the future (hypnotised) traces. We also learn, that exactly these two activities have proven to be extremely time consuming and error-prone and therefore very costly.

Having shown that recent scientific literature can be mapped well in the Tracing Activity Framework we will compare the TAF to other process models I have encountered in our survey in the next section.

Existing tracing process models

In this section I would like to introduce existing tracing process models. A process model consists of a defined set of activities which are executed in a specific order. Often they are connected to a specific approach for requirements tracing. In this case I will try to focus on process dimension but also introduce the approach very briefly. I will then highlight differences and similarities to our tracing activity model.

General information about traceability mechanisms is that they support the capture and usage of trace data, i.e. to document, parse, organize, edit, interlink, change, and manage requirements and the traceability links between them [112]. Ramesh and Jarke further report that most tools to not support the process of capturing and reuse of traces by guidance of enforcement in a systematic manner. Most of them offer just mechanisms for persistent storage and display of traceability information [112]. The limited process support is also mentioned in [81]. Nevertheless there exist some process models for RT.

Therefore we will compare them with our tracing activity framework to highlight similarities or differences. Recall the activities of our Tracing Activity Framework: Specification, Generation, Deterioration, Validation, Rework and Usage. First the author and type of model are introduced in each row. Then the phases of the model itself are shown briefly. In the last column the numbers are mapped to the TAF activities to see the overlapping.

Table 7 Existing tracing process models

Source	**Type**	**Model**	**TAF Overlapping**
Hayes et al. [56]	After-the-fact tracing	1. document parsing, 2. candidate link generation, 3. candidate link evaluation, and 4. traceability analysis	Generation (1,2) Validation (3,4)
Pinheiro et al. [106]	Generic model	1. In the definition phase classes of objects to be traced and of connections among them are defined in the project specification 2. In the registration phase, objects and relations are registered as the project evolves. 3. In the extraction phase traces are computed and presented using the models described later.	Specification (1) Generation (2,3)
Von Kenthen [81]	Generic model	1. Define Entities and Relationships 2. Capture Traces 3. Represent Traces 4. Maintain Traces	Specification (1) Generation (2,3) Validation & Rework (4)
Arkley et al. [9]	Generic model	1. Prepare Inputs for a Proposal: Capture and review the customer's requirements. 2. Manage, Analyse, Develop System Requirements: This phase includes developing the customer's new requirements. For each requirement a development risk and a verification method is identified and recorded. 3. Design: Recording of design decisions, this traceability information allows	Specification (1,2,3) Generation (4)

		change impact analysis later. 4. Prepare Test and Qualification Procedures: recording of traceability relationships between requirements and test procedures.	
Cleand-Huang et al. [22]	After-the-fact-tracing	1. Goal Modeling ->Specification • Construct Softgoal Interdependency Graph: elicit, analyse, negotiate, specify non functional requirements • Maintain SIG 2. Impact Detection • Link Retrieval: Probabilistic retrieval algorithm dynamically returns links between impacted classes and elements in the SIG. • User Evaluation: User assesses the set of returned links and accepts or rejects each one. 3. Goal Analysis • Contribution Re-Analysis: User modifies contributions from impacted SIG elements to their parents as necessary • Goal Re-evaluation: For each impacted element changes are propagated throughout the SIG to identify potentially impacted goals. 4. Decision Making • Decision: Stakeholders determine whether to proceed with the proposed change. • Impact Evaluation: Stakeholders evaluate the impact of the proposed change upon NFR goals, and identify risk migration strategies. The whole process is of iterative nature.	Specification (1) Generation (2) Validation (2) Rework (3) Usage (4)
Heindl et al. [63]	Value-based Requirements Tracing	1. Requirement Definition: Activity carried out prior to requirements tracing. 2. Requirements Priorisation: The input of this phase is a list of requirements, the output is the assignment of a precision level to trace for each requirement.	Specification (1,2,3) Generation (4) Validation (5)

		3. Requirements Packaging 4. Requirements Linking 5. Evaluation	
Gates et al. [49]	Constraint-based Requirements Tracing	1. Constraint and knowledge definition: Includes the definition of requirements and constraints in natural language. Constraints are manually linked to artifacts. 2. Requirement specification: in this phase the informal specification of requirements and constraints is transformed to a formal notation which is represented by means of a table. Here a distinction between ID, Event, Condition and Action is made. 3. Instrumentation: Involves analysing the flow of events in the code also by monitoring the code during its execution. The event specification is based on the events which were defined in the requirements specification. 4. Tracing and analysis: an initialize subsystem automatically determines and stores instrumentation points in a "path-tag table". This table shows which constraints are linked with which execution points.	Specification (1,2) Generation (3,4)
Huffman-Hayes et al. [70]	After-the-fact tracing	1. Identify each requirement 2. Assign unique identifier to each requirement 3. For each high level requirement, locate all matching low level requirements 4. For each low level requirement, locate a parent requirement in the high level document 5. Determine if each high level requirement has been completely satisfied 6. Prepare a report that presents the traceability matrix 7. Prepare a summary report that expresses the level of traceability of the document pair	Specification (1,2) Generation (3,4) Usage (5,6,7)

The frequency with which the TAF activities occurred within other consistent tracing process models:

- Generation 8 times
- Specification 7 times

- Validation 4 times
- Rework 2 times
- Usage 2 times

We see that the deterioration phase is missing. This may be due to the fact that it is an event and not an activity. Furthermore existing models are trace generation-centred as we have already learned. We will now switch to the presentation of experimental results to see how tracing approaches work in practice.

Experimental Results

Besides my general introduction of requirements tracing approaches, we also want to give a short overview, which results they are able to deliver. Due to very varying experiment setups it is hard to compare them in a structured way; we will therefore introduce each setup very briefly and mention afterwards the respective results.

To get a better idea about typical values of recall and precision a good table for assessment can be found in [70].

Table 8 Classification of Results [70]

Measure	Acceptable	Good	Excellent
Recall	60-69%	70-79%	80-100%
Precision	20-29%	30-49%	50-100%
Lag	3-4	2-3	0-2

General statements

Ramesh et al. [67] give some information on the effort of re engineering and redocumenting a system where no traceability information was recorded. The project was a flight control system with 75000 lines of code and 300 requirements. They state that the total effort for retrieving the original requirements was about 60 work months. The activities involved study of manuals, examination of code line by line and finally also hiring back engineers from the initial project.

Arkley et al. [9] report from their observed project that management was very pleased with the possibilities, traceability offered them. Among these were: a perceived reduction in project risk, better estimations of production costs and estimation of project progress (tracing for verification, accountability)

Effects for the customer

From Arkley et al. [9] we learn that the traceability system does not directly increase the costs for the customers. But he profits from the system as he gets a demonstration of how requirements related to his specification will be tested and what the progress of the entire project is.

Probabilistic vs. vector space model

Antoniol et al. [4] compare the performance of a probabilistic and a vector space model. They also report findings from two case studies. The first study was a freely available C++ library called LEDA 95 KLOC, 208 classes, and 88 manual pages. The aim was to map source code classes onto manual pages. Results where:

Table 9 Comparison of Probabilistic and Vector Space Model

	Probabilistic model				Vector space model		
Iteration	Retrieved	Relevant	Precision	Recall	Relevant	Precision	Recall
1	208	81	38,94%	82,65%	52	25,00%	53,06%
12	2494	96	3,84%	97,95%	98	3,92%	100,00%

In both cases again an increase in recall leads to a decrease in precision. The vector space model achieves 100 percent of recall sooner than the probabilistic model [4].

Heterogeneous Requirements Tracing

Form Cleland-Huang et al. [23] we get more detailed information about the costs of traceability. They provide examples for their cost estimation. The design parameters for the study were 500 requirements with an average of 5 links to establish for each requirement leading to a potential 2500 traceability links. Although these numbers are just estimates, the authors think that they are reasonable and provide a realistic analysis of the costs of each method.

Below is a table comparing the results of several different approaches for requirements tracing. The strategies were as following:

A: Coverage of 90% of the traceability relationship by manual generation of a traceability matrix

B: No formal traceability defined prior to the project

C: Coverage of the potentially most important traces (risky, volatile) by means of a traceability matrix

D: Manual coverage of 25% of traces, the rest is maintained with automated trace generation, definition of explicit trace strategy prior to the project.

Table 10 Heterogeneous Requirements Tracing

	A: Complete matrix coverage	B: No traceability coverage	C: Partial matrix coverage	D: Heterogeneous traceability
Explicit links	90%	0%	45%	25%
Total Cost of Traceability	$45,000	$33,750	$39,375	$25,125
Risk reduction	High	Low	Medium	High

Approaches A, C, D:

Prerequisites were that the average cost of establishing an explicit manual traceability link is $15 (calculated as 15 minutes of time at $60 per hour). The cost of maintaining a traceability link over 5 years is $5 on average. In case of Approach B: Prerequisite was that tracing is done and all changes are managed through brute force analysis. Each such analysis takes 1,5 hours also at a rate of $60 per hour. Another parameter is that 15% of the requirements change over five years.

Findings mainly were that heterogeneous traceability generates more costs at the beginning of the tracing process due to the risk assessment overhead. But later it offers a very cost effective solution as it simply leaves some uncritical requirements untraced or selectively traces medium-critical requirements with automated tracing approaches.

The second interesting information is, to see that the approach "no traceability coverage" also generates a significant effort. This is a good argument for traceability as it is sometimes perceived as a luxury extra in software development with no tangible benefit.

Vector similarity

Another comparison of traceability efforts can be found in the works of Hayes et al. [57] Over an existing dataset with 19 high level requirements and 50 low level requirements they compared the effort and the result quality of trace generation by an analyst and a vector similarity algorithm.

Table 11 Results of baseline algorithm compared to analysts

	Analyst 1	Analyst 2	Vanilla vector (10 x 10)	Vanilla vector (20 x 50)
Recall	23.0%	42.9%	23.0%	25.4%
Precision	15.0%	30.0%	17.6%	11.4%
Performance (min.)	65	150	seconds	Seconds

We see that in terms of speed the analysts are clearly outperformed by the algorithms. But it is also obvious that both results still need manual rework.

Value based requirements tracing

An approach for optimising the effort and benefit of requirements tracing is value-based requirements tracing (VBRT) Heindl and Biffl [63][36]. The main idea is to trace requirements with different granularity (precision) according to their respective value or risk, an idea also mentioned by Ramesh and Jarke in [112]. We also get information on how much time it takes to (manually) create traceability links in a matrix. For a project at Siemens where 46 functional requirements were specified and the following VBRT-phases were carried out:

Requirements Definition: We omit the effort generated in this phase as this activity must be carried out even without the implementation of requirements traceability.

Requirements Priorization: Involved stakeholders had to assess the risk and value of the traces. 50 minutes per stakeholder are mentioned as the required effort.

Requirements Linking:

- 45 minutes to trace a requirement at method level
- 10 minutes to trace one requirement at class level
- 2 minutes to trace a requirement at package level

These numbers are very interesting when comparing them with those of Cleland-Huang et al. who report to need just 15 minutes to set up a traceability link in a matrix [23]. It suggests that a coarser granularity has been employed in their approach right from the beginning.

The final result was VBRT took just 35% of the effort that would have been needed to trace all requirements with finest granularity. What's impressive about VBRT it is simplicity. No elaborate tool infrastructure is needed to achieve very good results. Furthermore if carried out at the beginning of the process, requirements prioritization not a very difficult task. An open question remains whether this approach would scale for much bigger projects, with say 300 requirements. As I have stated earlier this would be a reported limit for the application of traceability matrices.

Tracing to design models

Spanoudakis et al. also evaluated their approach [125]. Software-intensive TV systems, created by Philips, served as an example with the following preconditions:

- a document with 178 requirement statements expressed in natural language and organised through sections and subsections (e.g. installation procedure, audio functions, power

management),
- a use case specification document with 113 use cases, and
- an analysis object model composed of 108 classes, 70 attributes, 191 associations, and 277 operations.

Several different cases were executed. E.g. tracing specially the audio functions of the TV system or focusing on the video functions. Recall was found to be between 0.6 and 0.9 while precision also took these values but was inversely proportional. Generally precision was much higher than with probabilistic approaches, while recall dropped below 0,8 in some cases. So the analyst has to examine fewer false positives, but the coverage is worse to the other approaches.

Tracing to non-functional requirements
Cleland-Huang et al. report that SIG (Softgoal Interdependency Graph) traceability "resulted in recall metrics over 85% and precision of approximately 40 – 60%. The exceptions were the Cost SIG which performed exceptionally well with recall of 94% and precision of 81%, and the Availability SIG which performed rather poorly with recall of 50% and precision of 40%. Further experiments need to be conducted to more fully understand the implications of these results for individual SIG types, and on varied datasets." [22]

Tools
Although I focused on the process- or activity-side of traceability before the need for tool support is obvious. This is fortified by sources which state that:

"Requirements tracing is at best a mundane, mind numbing activity, as anyone who has spent any time performing this activity will tell you". [57]

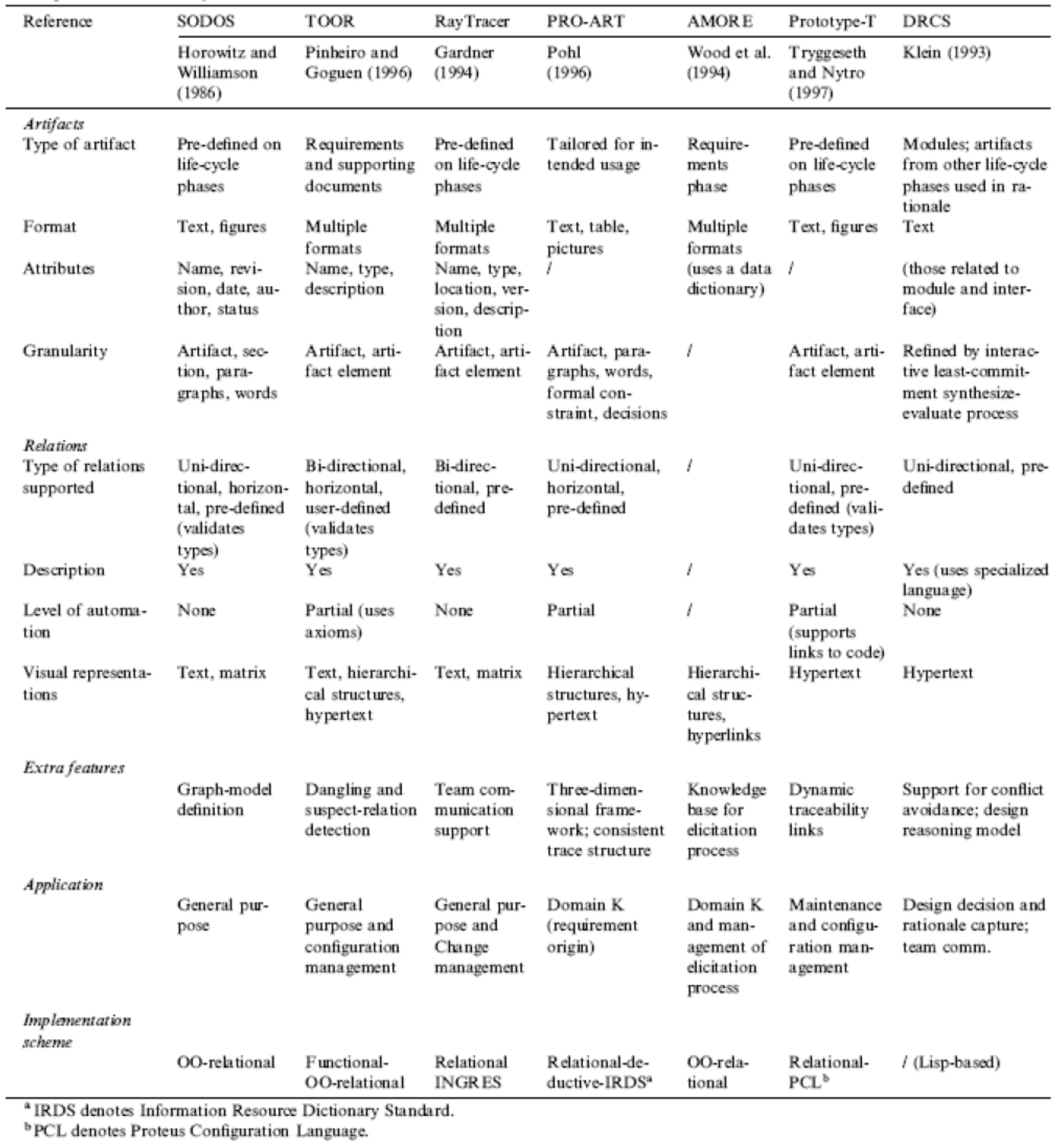
With the tools available in 1995 Ramesh et al. [67] state that the costs of implementing a traceability CASE tool could never be recouped in the initial project. They furthermore give some hints on the cost view of traceability: In their case study the costs for implementing traceability where claimed to be twice the documentation costs of the project. Actual costs were still larger than the estimation, but they were accepted for reduced total life cycle costs, higher quality and reduced maintenance costs. One of the first classifications for tools is given by Gotel and Finkelstein in [51]. I will introduce this classification below, try to give current examples and will extend the classification if necessary.

Table 12 Tracing Tool Classification

	General Purpose Tools	**Special Purpose Tools**	**Workbenches**	**Environments**
Description	include standard software such as word processors or hypertext editors	These tools explicitly support single activities of requirements tracing. An example is the KJeditor, which supports tracing between requirements and ideas	are somehow a collection of special purpose tools to support multiple activities of requirements tracing. They are built around a database system and should also accept a requirements document	additionally support all activities of software development. Besides requirements tracing they also offer teamwork, code management or other features. Today we are often referring to integrated development environments (IDE) as well
Examples	MS Office, OpenOffice, Webbrowser	Retro, TraceAnalyzer, TOOR,	RequisitePro, SLATE, DOORS,	Trace tools with interface to configuration- or requirements management system. JBuilder & CaliberRM

Another comparison of traceability tools can be found in [49]:

Table 13 Comparison of Traceability Tools



Reference	SODOS	TOOR	RayTracer	PRO-ART	AMORE	Prototype-T	DRCS
	Horowitz and Williamson (1986)	Pinheiro and Goguen (1996)	Gardner (1994)	Pohl (1996)	Wood et al. (1994)	Tryggeseth and Nytro (1997)	Klein (1993)
Artifacts							
Type of artifact	Pre-defined on life-cycle phases	Requirements and supporting documents	Pre-defined on life-cycle phases	Tailored for intended usage	Requirements phase	Pre-defined on life-cycle phases	Modules; artifacts from other life-cycle phases used in rationale
Format	Text, figures	Multiple formats	Multiple formats	Text, table, pictures	Multiple formats	Text, figures	Text
Attributes	Name, revision, date, author, status	Name, type, description	Name, type, location, version, description	/	(uses a data dictionary)	/	(those related to module and interface)
Granularity	Artifact, section, paragraphs, words	Artifact, artifact element	Artifact, artifact element	Artifact, paragraphs, words, formal constraint, decisions	/	Artifact, artifact element	Refined by interactive least-commitment synthesize-evaluate process
Relations							
Type of relations supported	Uni-directional, horizontal, pre-defined (validates types)	Bi-directional, horizontal, user-defined (validates types)	Bi-directional, pre-defined	Uni-directional, horizontal, pre-defined	/	Uni-directional, pre-defined (validates types)	Uni-directional, pre-defined
Description	Yes	Yes	Yes	Yes	/	Yes	Yes (uses specialized language)
Level of automation	None	Partial (uses axioms)	None	Partial	/	Partial (supports links to code)	None
Visual representations	Text, matrix	Text, hierarchical structures, hypertext	Text, matrix	Hierarchical structures, hypertext	Hierarchical structures, hyperlinks	Hypertext	Hypertext
Extra features							
	Graph-model definition	Dangling and suspect-relation detection	Team communication support	Three-dimensional framework; consistent trace structure	Knowledge base for elicitation process	Dynamic traceability links	Support for conflict avoidance; design reasoning model
Application							
	General purpose	General purpose and configuration management	General purpose and Change management	Domain K (requirement origin)	Domain K and management of elicitation process	Maintenance and configuration management	Design decision and rationale capture; team comm.
Implementation scheme							
	OO-relational	Functional-OO-relational	Relational INGRES	Relational-deductive-IRDS[a]	OO-relational	Relational-PCL[b]	/ (Lisp-based)

[a] IRDS denotes Information Resource Dictionary Standard.
[b] PCL denotes Proteus Configuration Language.

Parameters for Requirements Tracing

A detailed list of parameters relevant for requirements tracing is available in the appendix.

3.1.7 Lessons learned about systematic literature review

Systematic literature review offers many benefits for structuring the review process. Especially in the beginning of the review process it is helpful to exactly define and clarify the objective and constraints of the survey.

Concerning the conduction of the research process itself our experiences are variable. Using the data extraction form involves much overhead in writing the final report (as reported in literature).

When strictly sticking to the systematic literature review process, it must be clear that a smaller amount of "content" can be processed with the same time budget. For single researchers I do not recommend using the data extraction form (see appendix) it slows down the extraction process significantly and in the end extracted information is still scattered around many different documents. Furthermore one can often not copy and paste qualitative information (quotes) to the form as this might be possible for study results. So some rephrasing is necessary so that one does not need the initial paper any more. But when synthesising results from the data extraction forms and additional step of rephrasing might be necessary. I therefore discarded the data extraction form after having used it for 15 papers and started to use the final report as the form and experienced an improvement of the data extraction process. With this approach it is easier to relate extracted information to each other and to get a good structure after having extracted information after a few papers.

Using the data extraction form might be a good approach if many researchers are extracting information or if one just extracts quantitative information. The data extraction form also might be beneficial if one records results about a specific experiment and is not planning to write a report initially after the extraction. So the survey can be easily extended to further sources or updated with recent results without having hidden the results in a compact paper. Also another person could continue the research, so handing over the previous work is much easier.

Systematic literature reviews also imply some changes for the supervisor of a thesis or term paper. The supervisor must be familiar with the practice. He as well has to be involved in the development of the systematic review protocol and should be aware of changes, as the protocol steers the review process.

Open issues in requirements tracing

Often there is no process model behind tracing approaches, as they are very much generation-centred. Here researchers focus very much on the automated or after-the-fact generation of traces. Although researchers aim to achieve full automation in requirements tracing, I can summarise that there is currently no approach which accomplishes this. Although these approaches identify almost all relevant traceability relationships, a human analyst still has sort out 70% to 60% invalid traces after the generation.

The generation of less false positives is crucial for the success of these approaches. Surprisingly there are only very few approaches that deal with integrating trace generation functionality in the development environment. Some of these challenges might be addressed by the commercial tools for requirements management and are therefore probably not an issue for the scientific community. The most promising approaches in this field are those which fit into common development tools or configuration management systems. Other approaches might require designing a whole new requirements system, which is beyond the scope of one scientific work. So I can say that tools to support the continuous generation of traces always require developing plugins or interfaces for widely used systems.

Both manual and automated trace generation approaches must add additional support for the evolutionary nature of requirements. Only approaches like event-based tracing or ArchTrace offer support for automatic updates of traces. Building a valid RTM at the beginning of the project is just not enough as changes are very likely to happen (essentially they are the reason why we employ RT)

Another major shortcoming of current research is that I have only very limited empirical evidence about the behaviour of the proposed tracing approaches in real world conditions. Here standardised experiments especially for after-the-fact-tracing would help to justify the introduction of such approaches. I can say that manual trace generation approaches require the developer to invest a significant amount of time for traceability (Though not to use explicit traceability also takes time).

With automated trace generation approaches the effort is transferred to the manual validation phase as full automation cannot be accomplished at the moment.

So a good next step would be to develop a framework for experiments in requirements traceability. If accepted, such a framework would ease the comparison of different approaches and facilitate again the compilation of secondary studies. Providing standardised test-data could also be way to resolve the confusion how approaches actually performed. In such a way it is easier to calculate recall and precision as we know how many traces should be there. In this way more general empirical evidence, could highlight the most promising tracing approaches.

Finally the vocabulary varies a lot between the different publications. I highlighted this issue by stating the different synonyms in our tracing activities. I hope that the TAF can correct this in some way by introducing generally accepted terms so that we do not have to switch between the terms “link retrieval” “trace capture” and “trace generation” any more.

3.2 Model Building and Evaluation by Case Studies

Each research work starts with building a model representing the real world as good as possible. Thus, it consists of a lot of factors that influence each other. The basic steps of model building are:

- Identify the factors: the factors influence the approach to be evaluated and define the scope of analysis and evaluation
- Identify influences among factors: one factor often influences one or more other factors, which has an overall effect on the approach to be investigated.

The result is a model that contains all relevant factors that influence an approach to be investigated and is a basis for evaluation of this approach in empirical studies.

There are multiple types of empirical studies, e.g., controlled experiments, surveys, and case studies [48]. According to [48], they are used, when the effectiveness of approaches, techniques or methods is not clear. Empirical studies investigate the strengths and weaknesses of existing software engineering methods, techniques and tools. Empirical studies result in empirical knowledge or proven concepts. These will be added to a general ‘body of knowledge’ about software engineering theory. Concepts from this body knowledge will be used to solve new problems in software engineering practice. Empirical studies are therefore a driver to proceed in software engineering. According to [48], the high-level steps of an empirical study can be described as follows:

- *Study definition*: The objective of this step is to determine the goal of the study to be performed. Based on this goal and other factors described in later sections, an empirical strategy is to be chosen.
- *Design*: The objective of this step is to operationalize the study goal. Depending on the type of measurement data to be collected, the goal has to be expressed in a quantitative manner (including formal hypothesis on what to expect) when quantitative data is to be collected or refined into questions that are answered through interviews, questionnaires, or observation. In addition, appropriate data analysis methods have to be selected. The selection of data analysis methods has to take into account the kind of data (e.g., quantitative or qualitative, measurement scale) and the goal of the empirical study. Finally, the procedure for conducting the empirical investigation is devised and recorded in a study plan. This plan describes what needs to be done by whom and when.
- *Implementation*: The objective of this step is to produce, collect, and prepare all the material that is required in order to conduct the empirical study according to the study plan. Material to be prepared includes means for data collection (e.g., data collection forms, data collection tools, questionnaires, interview protocols) and other means such as experimental objects, e.g., documents to be inspected, systems to be modified. Usually, a pilot test (also called

trial run or pre-study) of the execution is performed in order to detect and correct any deficiencies in the prepared products or in the study design.

- *Execution*: The objective of this step is to run the study according to the study plan and collect the required data.
- *Analysis*: The objective of this step is to analyze the collected data in order to answer the operationalized study goal. The analysis is performed according to the data analysis methods selected during the study definition.
- *Package*: The objective of this step is to report the study and its result so that external parties are able to understand the results and their contexts as well as replicate the study in a different context.

Freimut et al. [48] define 3 different strategies for empirical studies, as listed in the following table:

Table 14 Definition of Empirical Strategies

Strategy	Definition
Experiment	A detailed and formal investigation that is performed in controlled conditions, i.e., we have control and can manipulate relevant variables directly, precisely, and systematically. The purpose of a controlled experiment is to make observations whose causes are unambiguous. This is achieved by isolating the effects of each factor from the effects of other factors to make significant claims of cause and effect.
Case Study	A detailed investigation of a single "case" or a number of related "cases". Such an investigation is performed in typical conditions, e.g., of a small number of typical, representative projects.
Survey	A broad investigation where information is collected in a standardized form from a group of people or projects.

The decision which strategy should be chosen mainly depends on the purpose of the study. The purposes of empirical studies can be divided into exploratory, descriptive, and explanatory. Explorative studies, which are often qualitative in nature, are appropriately tackled with case studies. Furthermore, case studies have the goal to investigate a "case" in realistic and representative conditions. That is why I used case studies to evaluate my research approaches: to have immediate feedback from real practitioners. The following sections explain my research approaches in detail. The evaluations in terms of case studies are also reported.

3.3 Chapter Summary

Systematic literature review offers many benefits for structuring the review process. Especially in the beginning of the review process it is helpful to exactly define and clarify the objective and constraints of the survey. Another benefit is that the setup of the review process is written down and can be reused by other researchers in order to update the review results.

Model building and evaluation in terms of empirical studies are a prerequisite to validate new approaches and methods. I mainly use case studies to evaluate my approaches in real project settings.

4 PLANNING OF DEPENDENCY MANAGEMENT

Managing dependencies in GSD projects brings multiple benefits, as described in section 2.1. On the other hand, if not planned well, capturing and management of dependencies causes considerable high efforts and creates tool challenges (which dependencies should be maintained with which tools?).

Thus, it is necessary to define at the beginning of the project, which data is useful for dependency management (based on parameters like the used project process, roles, tasks and artefacts in the project) and which dependencies are valuable to be captured and maintained, e.g., project managers should focus on high benefit and high risk dependencies.

Usually, a requirements engineer creates a document called "tracing policy" at the beginning of the project. This document contains the types of artefacts that will exist in the project and the types of dependencies between them that should be captured. Unfortunately, the reality very often breakes away from the plan, because it turns out to be too expensive to manage the dependencies as defined in the tracing policy.

As basis for planning dependency management activities in a GSD project, I developed a Tracing Activity Framework (TAF). TAF consists of tracing activities like trace generation, deterioratiation, validation, rework, and trace usage [60]. For each tracing activity, the relevant parameters that influence the activity are assigned. Thus, TAF is a good means to model requirements tracing alternatives by using the defined activities and parameters. At project start, the project manager can thereby precisely estimate the likely tracing efforts and benefits (such as reduced expected delay and risk) for a given set of dependency types.

I evaluated TAF with an initial case study in a large Austrian bank where I modeled 3 different tracing alternatives and used TAF to compare the efforts and the overall cost-benefit of each alternative in order to find the most suitable one for the given project. TAF can be used to calculate the number of traces to be established and maintained and supports the reasoning on which traces have priority and can be done with certain budget.

Tool selection is a crucial decision in early phases of a project and a prerequisite for my research contributions in the following chapters. I defined a value-based tool selection approach (in section 4.2) [66] that provides: (1) an overview on the tool needs from a practitioner's point of view, (2) an approach how stakeholder value propositions can be integrated into the tool selection decision in a simple and practical way, and (3) a discussion about which requirements tools (and features) have which value for certain types of projects.

4.1 The Tracing Activity Framework

The main goal of software development projects is to develop software that fulfills the requirements of success-critical stakeholders, i.e., customers and users. However, in typical projects requirements tend to change throughout the project, e.g., due to revised customer needs or modifications in the target environments. These changes of requirements may introduce significant extra effort and risk, which need to be assessed realistically when change requests come up, e.g., test cases have to be adapted in order to test the implementation against the revised requirements. Thus, software test managers need to understand the likely impact of requirement changes on product quality and needs for re-testing to continuously balance agile reaction to requirements changes with systematic quality assurance activities.

An approach to support the assessment of the impact of requirements changes is formal requirements tracing, which helps to determine necessary changes in the design and implementation as well as needs for re-testing existing code more quickly and accurately. Requirements tracing is formally defined as the ability to follow the life of a requirement in a forward and backward direction [51], e.g., by explicitly capturing relationships between requirements and related artifacts. For example, a trace between a requirement and a test case indicates that the test case tests the requirement.

Such traces can be used for change impact analysis: if a requirement changes, a test engineer can efficiently follow the traces from the requirement to the related test cases and identify the correct test cases that have to be checked, adapted, and re-run to systematically re-test the software product.

However, in a real-world project full tracing of all requirements on the most detailed level can be very expensive and time consuming. Thus, the costs and benefits to support the desired fast and complete change impact analysis need to be investigated with empirical data. While there are many methods and techniques on how to technically store requirements traces, there is very few systematic discussion on how to measure and compare the tracing effort and effectiveness of tracing strategies in an application scenario such as re-testing.

This paper proposes an initial *tracing activity model* (TAF), a framework to systematically describe and help determine the likely efforts and benefits, like reduced expected delay and risk, of the tracing process in the context of a usage scenario such as re-testing software. The TAF defines common elements of various requirements tracing approaches: trace specification, generation, deterioration, validation, rework, and application; and parameters influencing each activity like number of units to trace, average effort per unit to trace, and requirements volatility.

The model can support requirements and test managers in comparing requirements tracing strategies, e.g., for tailoring the expected re-test effort and risk based on the parameters: process alternatives, expected test case creation effort, and expected change request severity. I apply the TAF in a feasibility study that compares effort, risk, and delay of 3 tracing strategies: no trace reuse (NTR), full formal tracing (FFT) for re-testing, and value-based tracing (VBT).

The remainder of this chapter is organized as follows: section 4.1.1 summarizes related work on requirements tracing and requirements-based testing; Section 4.1.2 introduces the tracing activity framework and research objectives (section 4.1.3). Section 4.1.4 outlines the feasibility study and summarizes the results. Section 4.1.5 discusses the results and limitations of the study and lessons learned; a final section concludes and suggests further work.

4.1.1 Related Work on Requirements Tracing and Re-Testing

Several approaches have been proposed to effectively and efficiently capture traces for certain trace applications like change impact analysis and testing [4][36].

Many standards for systems development such as the US Department of Defense (DoD) standard 2167A mandate requirements traceability practice [113]. Gotel and Finkelstein define requirements tracing as the ability to follow the life of a requirement in both a backward and forward direction. Requirements traceability is an issue for an organization to reach CMMI level 3 making tracing an issue that many maturing software development organizations have to consider: the assessment for maturity level 3 there contains questions concerning requirements tracing: whether requirements traces are applied to design and code and whether requirements traces are used in the test phases.

The tracing community, e.g., at the Automated software engineering (ASE) tracing workshop TEFSE [36][41], traditionally puts somewhat more weight on technology than on process improvement. Basic techniques for requirements tracing are cross referencing schemes [44], key phrase dependencies [72], templates, RT matrices, hypertext [77], and integration documents [88]. These techniques differ in the quantity and diversity of information they can trace between, in the number of interconnections between information they can control, and in the extent to which they can maintain requirements traces when faced with ongoing changes to requirements.

Commercial requirements management tools like Doors, Requisite Pro, or Serena RM provide the facility to relate (i.e. create traces between) items stored in a database. These tools also automatically indicate which artifacts are effected if a single requirement changes (suspect traces). However, the tools do not automate the generation of trace links (capturing a dependency between two artifacts as a trace), which remains a manual, expensive, and error-prone activity.

Watkins and Neal [133] report how requirements traceability aids project managers in: accountability, verification (testing), consistency checking of models, identification of conflicting requirements, change management and maintenance, and cost reduction.

Gotel and Finkelstein [51] also state the requirements traceability problem, caused by the efforts necessary to capture and maintain traces. Thus, to optimize the cost-benefit of requirements tracing, a range of approaches focused on effort reduction for requirements tracing. In general, there are two effort reduction strategies: (1) automation, and (2) value-based software engineering.

1. Automation. Multiple approaches have been developed to automate trace generation: Egyed has developed the Trace/Analyzer technique that automatically acquires trace links based on analyzing the trace log of executing key system scenarios [39][40]. He further defines the tracing parameters: precision (e.g., traces into source code at method, class, or package level), correctness (wrong vs. missing traces), and completeness. Other researchers have exploited information retrieval techniques to automatically derive similarities between source code and documentation [4], or between different high-level and low-level requirements [70]. Rule-based approaches have been developed that make use of matching patterns to identify trace links between requirements and UML object models represented in XML. Neumüller and Grünbacher developed APIS [97], a data warehouse strategy for requirements tracing. Cleland-Huang *et al.* adopt an event-based architecture that links requirements and derived artifacts using the publish-subscribe relationship [23].

2. Value-based software engineering. The purpose of a value-based requirements tracing approach is not to reduce effort of each unit to trace (like automation) but to trace all requirements with varying levels of precision, and thereby reduce the overall effort for requirements tracing [63], e.g., high-priority requirements are traced with a higher level of precision (e.g., at source code method level), while low-priority requirements are traced with lower precision (e.g., at source code package level).

The effort used to capture traces should be justifiable with the effort that could be saved by using these traces in software engineering activities like change impact analysis, testing [35][69], or consistency checking. It is a matter of balancing agility and formalism to come close to an optimal level of cost-benefit [13]. The approaches described above serve the purpose of reducing the effort to capture traces.

$$\text{Effort for capturing tracing} + \text{effort for trace application by using traces} < \text{effort for trace application without using traces} \qquad \text{Eqn } (\mathbf{1})$$

Equation 1 captures this idea from a value-based perspective: to achieve a positive return on investment of requirements tracing the effort of generating and using traces should be lower than the effort for a trace application without traces. Such trace applications are (amongst others) change impact analysis and re-testing [47]. Besides effort of capturing traces, reduction of risk due to missed traces and delay due to the need to update traces are criteria that determine the usefulness of tracing approaches for software engineering activities.

Changes of requirements affect test cases and other artifacts [54]. Change impact analysis is the activity where the impacts of a requirement's change on other artifacts are identified [72]. Usually, all artifacts have to be scanned for needed adoptions when a change request for a requirement occurs. A trace-based approach relates requirements with other artifacts to indicate interdependencies. These relations (traces) can be used during change impact analysis for more efficient and more correct identification of potential change locations.

In [63] we proposed an initial cost-benefit model, where the following parameters that influence the cost-benefit of RT are identified: number of requirements and artifacts to be traced, volatility of requirements, and effort for tracing. In [61] we further discussed the effects of trace correctness as parameter influencing the cost-benefit of RT. However, tracing activities were not modeled explicitly, which would facilitate a more systematic discussion of the merits of different tracing approaches.

4.1.2 An Initial Tracing Activity Framework

Most work in requirements tracing research has focused more on technology than on processes supported by this technology to generate and use traces. For the systematic comparison of tracing alternatives I propose in this section a process model, the tracing activity framwork (TAF) [60], which contains the activities and parameters found in research tracing approaches; I label the framework as initial, although it is based on a systematic review literature and tracing activities in practice, as the external validation process has not yet concluded. The model can be used as basis to formally evaluate and compare tracing approaches as well as their costs and benefits.

Tracing Process Variants, Activities, and Parameters

The tracing activity framwork (TAF) in Figure 26 depicts a set of activities to provide and maintain the benefits of traces over time. The model is a framework to measure the cost and benefit of requirements tracing in order to compare several tracing strategies for a development project. The framework is based on previous work that identified tracing parameters, e.g., [23][36][37][41].

The activities in the model are building blocks identified from practice and literature and follow the life cycle of a set of traces.

Trace Specification is the activity where the project manager defines the types of traces that the project team should capture and maintain. For example, the project manager can decide to capture traces between requirements, source code elements, and test cases. This activity influences tracing effort based on the following parameters: Number of artifacts to be traced, number of traces, and artifacts to be traced. Other relevant parameters are tracing scope, precision of traces [36][41].

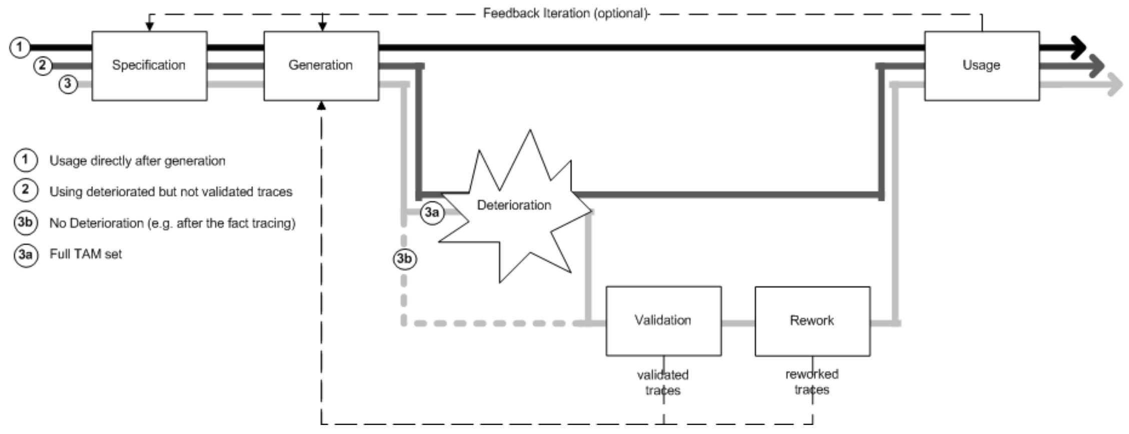

Feedback Iteration (optional)
1
2
3
Specification
Generation
Usage
1 Usage directly after generation
2 Using deteriorated but not validated traces
3b No Deterioration (e.g. after the fact tracing)
3a Full TAM set
3a
Deterioration
3b
Validation
Rework
validated traces
reworked traces

Figure 26 tracing activity framework: activities and process variants

Trace Generation. Trace generation is the activity of identifying and explicitly capturing traces between artifacts. Methods for trace generation range from manually capturing traces in matrices or requirements management tools that automatically create traces between artifacts based. The effort to generate traces in a project depends on the following parameters:

- Number of requirements: in a software development project; the effort for tracing increases with increasing number of requirements.
- Number of artifacts to be traced to the higher the number of artifacts, the higher is the effort to create traces between them.
- Average trace effort per unit to trace, which depends on the used tools and the point in time of tracing.

Other relevant parameters are: number of traces, tool support, and point in time of trace generation in the software development process, complexity/size of tracing objects, value of traces [63], correctness and completeness of traces.

Trace Deterioration. Trace deterioration is more the impact of external events than an activity. Traces can degrade over time as related artifacts change. If only the artifacts are updated, e.g., due to change requests, and the traceability information is not updated, the set of existing traces is likely to get less valid over time. Deterioration of traces affects the value of traces, because it reduces the correctness and completeness of traces.

Trace Validation and Rework. Trace validation is the activity that checks if the existing traceability information is valid or needs to be updated, e.g., identify missing trace links. In the example above, when artifact *A* changes fundamentally so that there is no longer a relationship to artifact *B*, trace validation would check the trace between *A* and *B* and flag it as obsolete. Trace validation is necessary to keep the trace set (traceability information) correct and up to date, so that the traces are still useful when used, e.g., for change impact analyses. I call the updating of traces "trace rework". Trace validation and trace rework are often performed together as they ensure correct and up-to-date traces and counter trace deterioration effects. The effort for validation and rework depend partly on the volatility of requirements.

The tracing activities are not necessarily performed in sequence. Furthermore, some activities are mandatory, like trace generation, whereas other activities are optional, as indicated by the arrows in Figure 26:

- *Trace Usage directly after generation (process variant 1 in Figure 26):* Trace deterioration depends on the changes made to certain artifacts. If traces stay valid over time and do not deteriorate, validation and rework are not necessary so that the existing traces can be used, e.g., for change impact analyses.
- *Using deteriorated traces (process variant 2 in Figure 26)* without validating and reworking them before is possible, but reduces the traces' benefits, because wrong or missing traces may hinder the supported activity more than they help
- *No Deterioration (process variant 3b in Figure 26):* Traces can be validated after generation whenever the project manager wants, even when they did not deteriorate.

Trace Usage. Finally, traceability information is used as input to tracing applications like change impact analysis, testing, or consistency checking [133]. The overall effort of such a tracing application is expected be lowered by using traces. The benefits of tracing during trace usage depend on parameters explained in [59].

The cost-benefit of requirements traceability can be determined as the balance of efforts necessary to generate, validate, and rework traces (cost); and saved efforts during trace usage, reduced risk and delay of tracing (benefits during change impact analysis). To maximize the net gain of requirements tracing the effort of generating, validating and reworking traces can be minimized, or the saved effort of trace usage can be maximized.

4.1.3 Research Objectives

The value of tracing comes from using the trace information in an activity such as re-testing that is likely to be considerably harder, more expensive, or to take longer without appropriate traces. If a usage scenario of tracing is well defined, trace generation can be tailored to provide appropriate traces more effectively and efficiently. Keeping traceability in the face of artifact changes takes further maintenance efforts.

The tracing activity model allows to formally define tracing strategies for a usage scenario by selecting the activities to be performed and by setting or varying the activity parameters.

I address the following research question:

- *RQ1: How useful is the TAF to model requirements tracing strategies and to determine and compare their efforts?*
- *RQ: To what extent can I balance the agility of a re-testing approach without using traces and the formalism of a systematic tracing approach for re-testing with a value-based approach?*

In order to evaluate the usefulness of the tracing activity model I conducted a small feasibility study in the finance domain, where I applied the TAF to 3 tracing strategies for the trace application re-testing. I discussed the usefulness of the re-testing strategies and the tracing model with development experts. If useful, the lessons learned from our evaluation could be a basis for extrapolation of tracing strategies and cost-benefit parameters to larger projects.

Re-testing is a software engineering activity that can be supported well by requirements tracing. The goal of a trace-based testing approach can be to make testing less expensive, less risky, and to reduce the delay. For a positive return on investment of tracing the effort to generate and maintain traces plus the effort of re-testing has to be lower than the effort of testing without tracing support.

4.1.4 Application of the TAF in an Industrial Feasibility Study

This section describes a feasibility study to validate the initial TAF framework concept. Together with practitioners from the quality assurance department of a large financial service provider I modeled 3 tracing strategies by using TAF building blocks and parameters and calculated tracing efforts of each strategy, their risks and delay in order to support the practitioners in deciding which tracing strategy provides the best support for re-testing in the practitioners' particular project context. This section describes an overview how I modeled each tracing strategy; detailed information of the study context can be found in the technical report [59].

The main focus of the study was to compare the efforts of each tracing strategy and the expected benefits of trace usage for re-testing. The TAF output variables were (1) the total effort of re-testing, (2) the risk of each strategy, and (3) the delay. Input variables were parameters covered the number of test cases, effort to create a trace, effort to create a test case, change impact analysis effort, etc. (see [59] for a comprehensive list of parameters).

Based on discussions with the experts in the industry environment and suggestions from literature I defined and compared 3 tracing strategies for re-testing: no trace reuse (NTR), full formal tracing (FFT), and value-based tracing (VBT). The data from this study can provide an initial snapshot in a typical scenario to find out whether the framework are useful to provide data and the proposed tracing strategies seem worthwhile for further discussion.

No trace reuse (NTR).

As baseline strategy I used the NTR strategy, which was the standard strategy in the feasibility study context; in this traditional re-testing process there is no trace support. Thus the activities of the tracing activity model are not performed and re-testing has to cope without traces: For each change request, the testers create new test cases instead of re-using and adapting existing ones. Obsolete test cases are replaced by new ones in order to avoid the risk of having redundant or inconsistent test cases, and to make sure everything is tested and test cases are still valuable after the change.

E(NTR) = #cr * #tc * tcn + dor. Eqn (**2a**)

E(NTR) = 20 change requests * 6 test cases* 1hrs + 6*700*8 min = 120 hrs + 560 hrs = **680 hrs**. Eqn (**2b**)

Equation (2a) calculates the overall re-testing effort following the NTR strategy: for each change request (#cr), new test cases are created with the expected effort (#tc* tcn). Finally, the testers have to check newly created test cases with existing test cases and delete redundant (obsolete) old test cases (dor).

In the particular study the total effort for NTR was as calculated in Eqn 2b (see [59] for detailed explanation.

Full formal tracing (FFT) for re-testing

In the FFT strategy, testers systematically establish traceability by relating requirements and test cases (full tracing) via a tool, the Mercury Test Director. When a change request occurs, they check, and adapt existing test cases whenever possible; else they create new test cases.

E(FFT) = #tntc * te + ciaFFT * #cr + tcnra * #tc * #cr Eqn (**3**)

Equation (3) calculates the overall re-testing effort following the FFT strategy: The formula consists of 3 parts: (a) upfront traceability effort (#tntc * te), which establishes traceability for each existing test case, (b) the effort to identify affected test cases for each change request (ciaFFT * #cr), and (c) the effort needed to either reuse (tcr) or adapt (tca) existing test cases, depending on the severity of the change requests (#cr). If existing test cases can neither be reused nor adapted, new test cases have to be developed (tcn).

The shares of test cases that can be reused, adapted, or need to be created anew typically has an important impact on the overall effort of re-testing.

The effort of FFT for change impact analysis depends on how many traces between requirements and test cases can be reused, or have to be adapted, or must be created. These values depend on the type of change request, as not every change request effects artifacts in the same way, e.g., there are simple low-effort change requests, e.g., affecting locally the user interface, whereas more severe change requests may need more extensive adaptations in several software product parts. Eqn 4a and 4b depict the efforts for FFT.

CIAFFT effort overall = 54 + 86 + 33 hrs = 173 hours Eqn (**4a**)

E(FFT) = upfront trace effort + CIAFFT = 350 + 173 = **523 hrs** Eqn (**4b**)

Based on effort reports for typical change requests in the case study context I categorized change request into the classes: Mini (small), Midi (medium), and Maxi (severe) (see [59] for details).

Value-based tracing (VBT)

VBT is a hybrid between FFT and ad-hoc tracing. Usually the upfront effort for FFT is considerably high, because all existing requirements have to be traced to test cases. VBT tries to reduce this tracing effort by establishing traceability on a coarse level (to test case packages instead of particular test cases) and to refine them ad-hoc when necessary. That means that all requirements are traced to test case packages and when change requests occur for some requirements, the traces from these test cases are refined to particular test cases to improve change impact analysis. Equation (5) calculates the overall re-testing effort following the VBT strategy:

$$E(VBT) = \text{upfront trace effort (on package level)} + \text{change impact analysis (VBT)} \qquad \text{Eqn } (\mathbf{5a})$$

$$\mathbf{E(VBT)} = 70 \text{ hrs} + 325 \text{ hrs} = \mathbf{395\ hrs} \qquad \text{Eqn } (\mathbf{5b})$$

The upfront tracing effort for VBT is lower since traces have to be captured on more coarse level of detail than with FFT (70 hrs in comparison to 350 hrs with FFT). The change impact analysis effort for VBT consists of refining traces from changing requirements to the affected test case packages. The effort for identifying particular test cases by refinement was 325 hrs in the study resulting in a total effort of 395 hrs for the value-based tracing strategy to support re-testing.

4.1.5 Discussion

The purpose of the case study was to evaluate the feasibility of the TAF to model tracing strategies, in our case with focus on effort, also considering delay, and risk.

For practical reasons, the case study size and context was chosen to allow evaluating the approaches in a reasonable amount of time. However, the case study project setting seems typical in the company and financial service sector; the project context allows reasonable insight into the feasibility of the trace-based re-testing strategy in this environment.

In the feasibility study project, I deliberately applied a simple process variant from the TAF focusing on the activities trace specification, trace generation and the usage of generated traces for re-testing. Trace deterioration, validation and re-work were not enacted; rather I assumed for trace usage all generated traces to be correct. While this reduction of scope limits the experience this focus was found beneficial to make sure that the proposed process is actually applied in the practical setting.

As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings. Furthermore, I analysed only a simple instantiation of the tracing activity model in the case study; consisting of trace generation and trace usage, but without considering trace deterioration, and consequently neither trace validation nor rework. In practice incorrect traces and trace deterioration can considerably lower tracing benefits and need to be investigated.

Modelling the 3 tracing strategies by using TAF activities and parameters provided data points for effort of each strategy. As these are single data points in a specific study setting, I see the results as snap shots, which should motivate further data collection to allow statistical data analysis and sensitivity analysis.

Comparing the 680 person hours effort of the NTR strategy, where new test cases are created for each test case, with the FFT alternative, with 523 person hours, FFT takes around 20% less effort. In this case the upfront investment into traceability pays off. In many cases, full tracing (tracing each requirement to each relevant test case) can cause considerably high effort which may prevent tracing in practice. Here, the study results suggest that the value-based strategy to trace requirements to test cases on a coarse level and refine them later on demand to be a promising approach that can significantly save efforts.

Besides effort, the alternatives also differ in delay when traces can be used for the trace application, in our case re-testing. VBT has a larger delay, because trace refinement has to be done before re-test. Concerning risk, NTR would be more risky if obsolete test cases were not checked; Inconsistent or redundant test case sets could then result in increased hidden testing effort or lower-quality test sets.

Lessons Learned from the Feasibility Study

The tracing activity model was found useful for systematically modeling the tracing alternatives, e.g., no tracing, systematic full tracing, and value-based tracing for the certain tracing application re-testing. The model helps make alternative strategies comparable, as it makes the main tracing activities explicit and allows mapping relevant parameters that influence tracing costs and benefits. Some input parameters (like number of change requests in the project, or effort to create a test case) had to be estimated based on practitioners' experience. Other data elements could be measured in the project context, e.g., number of requirements. The TAF allows choosing from the listed tracing activities and parameters and selecting the relevant ones to model tracing strategies for a particular usage scenario.

According to the expert feedback the calculated efforts provide a good input to reason about which tracing strategy seems most beneficial in a particular project context.

The lessons learned of our study for trace-support change impact analysis are:

- TAF provides useful building blocks for reasoning about relevant parameters (efforts, risks, etc.) of a tracing strategy and estimation of outcomes in advance helps to rationally discuss candidates for the best-fitting strategy.
- The volatility of traces is a major risk for full tracing. In volatile parts of the project, agile (just in time) or value-based approaches are favorable as full tracing has a particularly high risk of loosing upfront investments in tracing in these volatile areas.
- Full tracing provides detailed traces, which are particularly useful for situations when artifacts are not volatile and quick feedback is at a premium, e.g., for comprehensive cross checks at milestone reviews.
- If calculated efforts of tracing strategies do not differ significantly, choose a value-based strategy to provide full (complete) traceability at a coarse level of detail. This coarse-level traceability can than be refined on demand with reasonable total effort for change impact analysis.

4.2 Value-Based Selection of Requirements Engineering Tool Support

Software development projects in big software development companies span a wide range of types, differing in domain, project size and duration, degree of distribution, application domains, tooling and process models.

Handling requirements in such projects is complex due to the following factors:

- Interdependencies of requirements to various other artefacts like design specifications, source code, test cases;
- Requirements management spans the whole product lifecycle and has to interact with other

disciplines (from design to maintenance);

- All project members are involved in handling re-quirements.

Project managers and requirements engineers, as well as other project participants, need appropriate requirements tool support to manage this complexity. There is a wide range of requirements tools available with very different capabilities, concepts, and termi-nologies. Therefore, a challenge for the project manager is to evaluate and select the most appropriate tool.

In this context, the Support Center for Configuration Management (SCCM) at Siemens IT Solutions and Services PSE (SIS PSE) developed a comprehensive and easy-to-use requirements tool selection approach that aims to help project managers in a value-based way to select requirements tools. The SCCM is a unit in the SIS PSE providing (amongst other services) consulting for employees and enabling experience ex-change among them on key topics like configuration and requirements management.

In this subsection, I present an approach that is based on a generic requirement engineering (RE) process description that considers RE as an intrinsic part of the whole development processes (rather than an isolated discipline). From this process description I derived a well-structured requirements tool feature catalogue. Additional input came from literature and empirical information from experienced tool users at SIS PSE. This feature catalogue serves two purposes: (1) it allows a standardized rating of tool capabilities in order to com-pare different tools (tool rating model), and (2) it can be used to create a value-based profile of needed tool capabilities for a given project situation (tool value model).

The feature catalogue can conveniently be repre-sented as feature tree, consisting more than 80 features. Top nodes in the tree (derived from the RE process description) are similar to the structure of the require-ments analysis part of the guide for the software engi-neering body of knowledge (SWEBOK) [14].

Compared to already existing feature catalogues like INCOSE [71], new essential requirements tool features were identified. Furthermore, essential requirements, originating in our view on RE as tightly integrated with other involved disciplines (from design to mainte-nance), were given respect. With this approach, I conducted an initial tool evaluation with 5 commercial and 2 in-house requirements tools.

4.2.1 Value-Based Tool Selection Approach

The main parts of the approach are: (1) the tool rating model that was created by developing a generic requirements engineering process (step 1), deriving a tool feature tree from the process (step 2), and finally an evaluation of requirements tools (step 3), and (2) the value rating model, that allows project managers to rate the value contribution of requirements tool features for their project.

Research Questions

The research questions I address are:

What are the tool needs from a practitioner's point of view?

I developed a practice-oriented tool feature tree with experienced tool users at PSE.

How can I integrate stakeholder value propositions into the tool selection decision in a simple and practical way?

I describe a tool rating and value rating model as means to support a value-based tool selection.

Which requirements tools (and features) have which value for certain types of projects?

I discuss the proportion between tool features and their value for certain project types.

Developing the Tool Rating Model

The approach to develop a rating model was thought as follows: First, the need for tool support of requirements engineering activities should be modelled without respect to the current abilities of commercial requirements tools.

The modelled need resulted in a tool feature tree that was then used as checklist to evaluate how existing requirements tools meet the needs for requirements tool support. The following paragraphs describe the approach in more detail.

Step 1. Development of a requirements engineering process description

This process describes the main activities in requirements engineering, e.g., requirements elicitation, requirements validation, requirements management. Well knowing that the requirements tools to be evaluated focus on requirements management and do not support requirements elicitation (most requirements tools used in SIS PSE are requirements management tools), the evaluators still modeled the need for requirements elicitation tool support to raise the full scope of requirements engineering tool support needs.

Table 15 Illustrating subset of new tool feature needs elicited at PSE survey and interviews

Tool feature	Description
Definition of a workflow for requirements	A workflow (states, roles, state transitions) is configurable for requirements, like it is usual for other artifacts in configuration management.
Automated generation of bi-directionality of traces	When the user creates a trace between artifact A and artifact B, it automatically establishes a backward trace from B to C without interaction of the user.
Definition of user-specific trace types	An authorized user can define trace types and assign names to these trace types .
Suspect traces	When a requirement changes, the tool automatically highlights all traces related to this requirement for checking and updating traces.
Long-term archiving functionality	All data in the tool can be archived in a format accessible without the tool in order to re-setup the environment if necessary.

Step 2. Development of a tool feature tree based on the RE process description

For each requirements engineering activity the evaluators derived desirable tool features to support the activity. Additional input for this step came from INCOSE's tool feature list, and from experienced tool users at SIS PSE (see elicited new tool features in table 1). Features are described in an understandable and unambiguous manner (uses cases or scenarios were attached where appropriate).

The result was a tool feature tree structured similarly to the elements of the “software requirements analysis” part of the guide to the software engineering body of knowledge (SEWBOK) [14]. The leaves of the feature tree consist of more than 80 tool features. A cutout of the feature tree is depicted in Figure 27. The software requirements analysis module of the guide to the SWEBOK contains elements like requirements engineering process, requirements elicitation, requirements analysis, requirements validation, and requirements management. For the latter element, Figure 27 depicts a sub-tree containing the requirements tracing-related tool features.

The nodes below “requirements tracing” are a classification layer that supports clarity and togetherness of tool features. For example, the features under “types of traces” represent a tool’s ability to create traces between requirements and design, source code (at different levels like class or method level). “Support of understandability” refers to the ability to name traces. “Trace generation” contains features that allow manual or automated trace generation. “Automated generation of bi-directionality for traces” means, that a tool is able to automatically establish a trace from B to A when the user creates a trace from A to B.

“Display traces” contains features that support different ways of displaying traces: “trace representation” contains graphical, matrix, or tabular trace representation; “suspect traces” means that the tool highlights traces (suspect traces) that have to be checked, because one artifact attached to these traces has changed.

Besides requirements management, the feature tree also contains features for requirements elicitation, validation, documentation (e.g., import and export features), requirements engineering process configuration, and configuration management, as well as usability features.

The advantages of the tool feature tree in comparison to other types of tool feature catalogues are:

- The feature tree addresses the whole requirements engineering process, thus, not only on requirements management;
- The tree is structured similarly to the SWEBOK to provide a clear framework for software engineers,
- The tree contains new features that came from experienced tool users at PSE (see table 1 for examples).

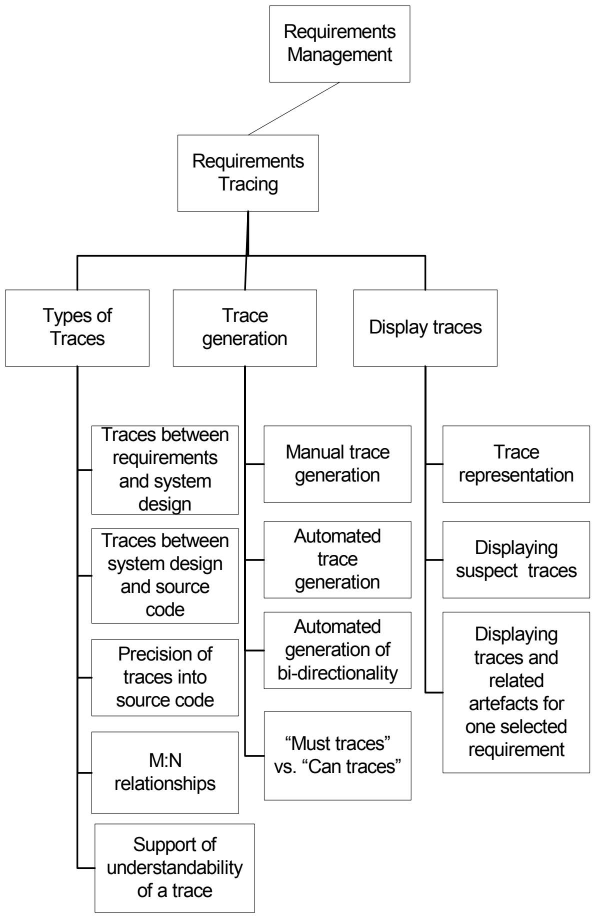

Requirements Management
Requirements Tracing
Types of Traces
Trace generation
Display traces
Traces between requirements and system design
Traces between system design and source code
Precision of traces into source code
M:N relationships
Support of understandability of a trace
Manual trace generation
Automated trace generation
Automated generation of bi-directionality
"Must traces" vs. "Can traces"
Trace representation
Displaying suspect traces
Displaying traces and related artefacts for one selected requirement

Figure 27 Cutout from the feature tree

The feature tree was found to be mostly stable; however with changing software engineering and requirements engineering processes, some parts of the feature tree are expected to evolve.

Step 3

Support Center Configuration Management at SIS PSE performed initial tool evaluations of 5 commercial and 2 in-house requirements tools.

The elements of the feature tree (leaves) were used as a checklist for the evaluation and for a given tool each feature was rated on a scale from 0 (feature not provided by the tool) to 4 (feature fully provided by the tool). Additionally, comments of how each feature was implemented in a tool were captured for each feature. The rating of an example tool is illustrated in Table 16. The results of our rating model were validated with experienced tool users at SIS PSE.

As mentioned above, the evaluators in the Support Center evaluated seven tools. The tools were selected, because either they were already in use in the PSE, or there were at least negotiations conducted with tool suppliers. The set of evaluated tools contained some of the most popular requirements management tools, as well as PSE in-house solutions. The rating model is scalable by rating new requirements tools with the feature tree as checklist.

Table 16 Example rating of tool features

Tool feature	Rating Tool X	Comment
Nameable trace types	0	Only pre-defined trace types provided.
Automated generation of bi-directional traces	4	When a trace from A to B is captured, the tool automatically creates an inverse trace from B to A.
Tracing into code	2	Possible only to code files, e.g., java files, no traces to method level possible.

The resulting artifacts of the rating model were: a feature tree outlining desirable functionality for requirements-related activities, and for each evaluated tool a rating form (excel) describing if and how the tool provides a feature in order to provide means for comparison of existing tools.

4.2.2 Tool Value Model

As feature catalogues are mostly value-neutral, this subsection describes, how project managers and leaders of organizational units can use the feature tree described above to value the tool features and thereby find the most suitable tool that best addresses the project- or unit-specific stakeholder value propositions.

First of all, the rating model above provides a simple means for project managers to get an overview about features of existing tools. It is a good means to point out the importance of certain features which project managers were unaware of. Further, project managers have two options to rate the value of a given tool: a combined rating and a rating by value classes, as described in the following paragraphs.

Combined value rating

The combined rating allows a project manager to rate the value of features with the following scale: A(3 points), B(2 points), C(1 point), D(0 point). An example is depicted in Table 17.

Table 17 Examples for combined value rating model

Nr	Tool feature	Value rating
1	Nameable trace types	D
2	Automated generation of bi-directional traces	A
3	Tracing into code	A
4	Displaying suspect traces	B

In the example, the project manager rated “nameable trace types” with D (0 points) because in his project he does not want to create individual trace types; he rather wants to use existing trace types. “Automated generation of bi-directional traces” is very valuable (A), because the project is about a very large system and lots of traces have to be generated. Therefore, the project manager supposes this feature to save efforts for requirments tracing. “Tracing into code” is demanded by internal instructions and is also rated very important. Highlighting traces that have to be checked due to the change of an attached artefact was considered as medium important.

Adding up the points of this rating, results in a total score of 8 points. Subsequent normalization in the example shows that feature 1 provides 0% of the tool's value for the project manager, features 2 and 3 each provide 37.5% value, and feature 4 provides 25% of the value. Starting from these results, the project manager, assisted by Support Center members, can map these values to the tool feature ratings and identify the best-fitting tool.

A weak point of this kind of combined rating is that A, B, C, and Ds are lumped together in terms that one A is worth 3 Cs, and one B is worth 2 Cs. Therefore, another option for value rating is the rating by value classes.

Rating by value classes

Instead of using a quantitative approach for the value rating, rating by value classes enable a qualitative way of reasoning. The value classes are not asigned with numerical values, but with qualitative values, namely: A (crucial), B (important), C (nice to have), D (unimportant). Thereby, the interdependencies between the rating classes are cancelled.

Again, a project manager can now rate each feature of the feature tree with A, B, C, or D. Finally, the numbers of As, Bs, Cs, and Ds can be counted separately, and the tools' numbers can be compared to find the best-fitting tool. Besides this feature-based value rating, which represents a bottom-up rating, the feature tree further allows a top down rating, where the high-level nodes, e.g., the requirements engineering main activities are rated first to eliminate unimportant sections of the feature tree, and then the leaves of the remaining tree are rated as described above. This saves effort, because parts of the feature tree can be blended out because of their unimportance, which is another benefit of the feature tree structured according to the requirements engineering activities.

In addition to value rating of already evaluated requirements tools, a project manager has also the option to rate his usual requirements engineering tool support using the tool feature list. This gives him the opportunity to compare the proposed tools to his usual tool and allows weighing the effort for adopting a new tool against the tool support improvement.

4.2.3 Discussion: Value of tools/features for Project Types

A coarse analysis of the project types at Siemens PSE showed us the big variety of project types ranging from small, collocated projects with a handful of members, agile projects, to large, highly-distributed projects with dozens of project members.

These projects do not only differ in organizational aspects, but also concerning topics. Besides many others, there are traditional software development projects, covering the whole software development lifecycle, as well as pure verification projects, where the main focus is on system testing and ensuring traceability between requirements and test cases. While the value of a traceability feature that enables traces between requirements and test cases may be only of average importance for the traditional software development project, it has a great value for the verification project.

This leads to the assumption that the value of tool features can not be stated independent from the project context and the stakeholder value propositions within this project. Based on this assumption, our value-based tool selection approach provides a means to identify the stakeholder value propositions by developing a value model and thereby find the optimal tool support.

Furthermore, our feature tree does not only represent features that are implemented in available requirements tools, but also features that are desireable for practitioners. For example, practitioners wished to have forums withing their requirements tools to discuss and negotiate requirements in distributed projects. Until now, neither of the evaluated tools provided these features. Thus, the feature tree, if maintained properly in the future, could also be a means for tool vendors to identify needs for new features.

In this context, it is also a means for the Support Center to explain features to a project manager and to point out the value a feature might have for him, which he was unaware until then.

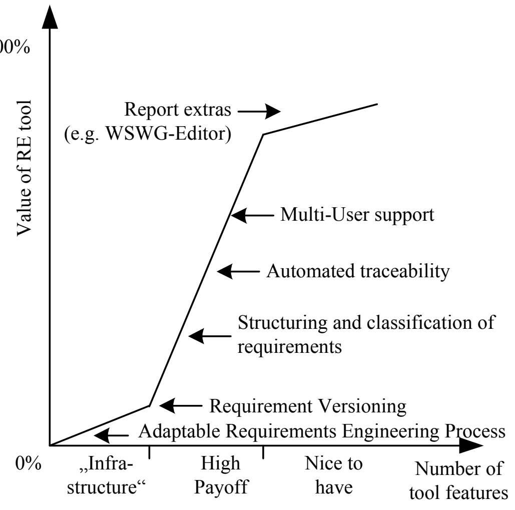


Figure 28 S-curve to illustrate the proportion between features and value

Another discovery I made during developing the tool selection process was that the proportion between the number of tool features of a tool and the total value of this tool seems not to be linear. I suppose it to be S-shaped, as exemplarily depicted in Figure 28. This figure is an estimate for the value of some examplary tool features for a large-scale, highly-distributed project, and is just for illustration. It needs further work to more precisely analyse the proportion between features and value.

The segments of the S-curve are "infrastructure", containing necessary tool features that do not provide a specific value. The middle part is the "high-payoff" segment containing features that in context of the given project type provide a strong increase of value (steep value ascent). The last segement is the "nice-to-have" part that contains features that do not extremely increase the total value of the tool.

In our example in Figure 28, an "adaptable requirements engineering process " feature means that a state model for requirements can be configured in a tool, containing permissions which tool users are allowed to change the state of a requirement. Possible states could be: elicited, validated, to be checked, etc. In Figure 28, this feature provides only a low value for the project (see the low ascent of the curve), because it provides a good infrastructure, but could also be realized outside the tool by carefully defining a process. "Requirements versioning" is also something like a "basis requirement" that does not provide a specific value to the given project type.

The features in the high-payoff segment all provide an increased value for the given project type. Structuring and classification of requirements, e.g., by a requirements tree structure or a file explorer structure, helps to keep the overview, which is very valuable due to the high amount of requirements usually existing in large-scale projects. Although one would expect that this is a feature that is provided by all existing tools, there are some that do not provide this feature. Thus, the latter would not be a good solution for the large-scale project type.

Due to the high number of requirements and artefacts emerging during the project, some kind of automated traceability support, e.g., automated generation of bidirectional traces, are very valuable because they help to save a lot of effort. Since all traces have to be birectional, in order to follow the life of a requirement in a forward and backward direction, such a feature halves the effort for bidirectional traceability, because only a trace in one direction has to be created. Multi-User support in highly-distributed projects, where one team is e.g. located in China, one in Austria, and one in Slovakia, is extremely valuable in order to support coordinated collaboration, file access, and well-defined user permissions. Furthermore, multi-project capability can be an inevitable requirement for organization-wide used solutions. The last section of the S-curve in our example contains features that do not provide a big additional value, illustrated by the flat ascent of the curve. Such features are all kind of add-ons, e.g., additional report functionalities or WYSIWYG-editors.

A further goal will be to more precisely analyze the project types and its characteristics and to map tool features to these project types in S-curve manner in order to reflect the value of certain features for certain project types (see further work). Further work also contains ways how to evaluate the value of features for project types.

4.2.4 Conclusion

There is a big variety of projects at SIS PSE and project managers or leaders of organizational units face the challenge to select their requirements tool that best fulfills the project- or department-specific (one solution for many projects) needs. This is a challenge because the number of available requirements tools, commercial as well as open source tools, is very high, and the comparison of these tools is very expensive and time-consuming.

For that reason, the Support Center Configuration Management, a unit of the SIS PSE competence basis department, developed an approach to provide decision support for project managers in requirements tool selection decisions.

The approach contains:

- A requirements engineering process description that provides an introduction and structuring of requirements engineering activities (requirements elicitation, analysis, documentation, validation, prioritization, management)
- Based on this process description, evaluators derived a practice-oriented feature tree, which is structured after the guide to the software engineering body of knowledge. The structure provides clarity and a good overview.
- The evaluators used this feature tree as a checklist to evaluate seven currently available requirements tools. The result is a rating model by which the strength and weaknesses of each tool can be easily compared.
- Furthermore, a value model allows project managers to value the importance of the features for their projects and thereby identify the most suitable requirements tool support that best addresses the project manager's value propositions.

The resulting artefacts are a good means for the Support Center to provide decision support to project managers and other tool decision makers. The rating model can be easily extended by rating new tools. It helps to point the value of certain features out that the project manager was not aware before.

The long-term goal could be to reduce the number of used requirements engineering tools, and thereby to provide a higher level of support for a given tool. Furthermore, costs for tool support and maintenance could be reduced.

The initial feasibility study showed that the proportion between the number of tool features and value is not linear, but s-shaped. Furthermore, different tool features seem to provide different values to different kinds of projects, which will be analyzed in further work.

4.2.5 Further Work

Further work will be the extension of the existing rating model (the currently evaluated tools) with other requirements tools, open source as well as other commercial requirements tools. This should ease the comparison of tools on feature basis.

Furthermore, I aim to concretize the classification of project types in order to improve the mapping between tools and tool features to project types and to further analyze the proportion between tool features and value for these defined project types.

In a last step, I want to develop some kind of tool support that: (a) allows evaluators to rate new tools and to store and update their ratings for given tools, and (b) supports project managers, assisted by Support Center representatives, in more efficiently finding the most suitable requirements tool. It is also planned to develop a requirements management experience base with the tool rating results as input.

4.3 Chapter Summary

In this chapter I proposed an initial tracing activity framework (TAF) as framework for defining and comparing tracing strategies for various contexts. For each tracing activity, relevant parameters were identified from related work and practice and mapped into the model. The model allows to systematically compare tracing strategy activities, their costs and benefits. I performed a small study in the financial service domain, where I evaluated the feasibility of the tracing activity model.

Main results of the study are: a) The model was found useful to capture costs and benefits of the tracing activities to compare the different strategies; b) for volatile projects or project parts just-in-time tracing seems favorable; c) for parts that need quick feedback detailed upfront preparation of traces can be warranted; d) a combination of upfront tracing on a coarse level of detail (e.g., package or class level) and just-in-time detailed tracing of really needed traces can help balancing tracing agility (for use in practice) in a formal tracing framework (for research and process improvement). Further work will be to use the TAF a) as a framework for a systematic literature review for requirements tracing literature and b) to apply the TAF for studies on tracing strategies in other contexts.

Tool selection decisions should be performed with respect to the stakeholders’ value propositions, in order to meet the expectations that tool users have from a tool. The value-based tool selection approach contains a feature tree that provides a good overview on tool features. The initial feasibility study showed that it helps decision makers to get a feeling which tool features are actually provided by requirements engineering tools.

5 CAPTURING DEPENDENCIES EXPLICITLY, SYSTEMATICALLY AND CONSISTENTLY

In the last decades requirements tracing approaches have emerged with the goal to explicitly capture and model dependencies between requirements and artefacts like design, source code, and test cases as traces (also called trace links in this chapter).

The main drawback of manual tracing approaches is the high effort, which in practice leads to (a) rather sporadic and unsystematic trace capture (which in turn often leads to insufficient quality of traces for dependency analysis) and (b) high effort to motivate personell to capture traces (which in turn leads to delay and high costs, and again bad trace quality, if motivation is not too high).

Thus, research approaches emerged to automate trace generation in order to reduce tracing effort. Up to now, these approaches continue to suffer from (a) low trace quality, and (b) insufficient integration of tool support for tracing; so available information on dependencies in data repositories of different tools is hard to access and integrate for dependency analysis.

As mentioned in the introduction, I tackle these issues for managing technical dependencies with the following approaches:

(a) *Value-based requirements tracing (VBRT)* systematically supports project managers in tailoring requirements tracing precision and effort based on the parameters stakeholder value, requirements risk/volatility, and tracing costs to achieve high-benefit traceability information with limited budget. VBRT is described in the following subsections and the following questions were analyzed:

- Section 5.1 contains a case study of VBRT [63] and focuses on the research question: To what extent can VBRT reduce requirements tracing efforts?
- Section 5.2 and 5.3 focuse on the investigation of (a) the level of detail of traces among artifacts (package, class, method levels); (b) the value of the artifacts that are traced (high-value artifacts justify a higher level of tracing effort); and (c) the points in time of trace generation (early vs. late) [64][41][36]. The simple question I asked was whether increases in quality of trace links do justify their cost.
- Section 5.4 analyzes the question: How does the correctness of requirements trace links influence the value of traces for change impact analysis [61]?

(b) *Integrated Developer Tool support for dependency management* reduces the effort for capturing dependencies between requirements and source code elements significantly by providing semi-automated support for users and thereby improves trace quality such as correctness and completeness of captured traces [130]. Furthermore, traces have to be (re-)checked after a change and this approach also supports trace maintenance during system evolution. Integrated developer tool support is described in section 5.5

5.1 A Case Study on Value-Based Requirements Tracing

Software development includes the production of various types of artifacts, e.g., requirements specification documents, architecture descriptions, source code, and test cases. These artifacts provide different views on the system at different points of time. It is obvious that these artifacts do not exist in isolation from each other. Instead, they are related to and affect each other, e.g., if one requirement in the requirements specification document changes, other documents often have to be changed in order to preserve consistency. Furthermore, these artifacts typically evolve to some extent concurrently during development.

Requirements tracing is the ability to follow the life of a requirement in a forward and backward direction [51]. In the software development context, requirements tracing has important benefits, e.g., capturing traces in weekly intervals during development can support developer teams in keeping an overview on which requirement is implemented where in the source code. Project managers aim at keeping track of interdependencies between various artifacts of the software development lifecycle to find out potential requirements conflicts, to better understand the impact of change requests, and to fulfill process quality standards, such as CMMI requirements [102].

In literature approaches like [39] [72] [106] support requirements tracing activities like identification of requirement conflicts, change management and impact analysis, release planning, program comprehension, model consistency checking, and testing (verification and validation). However, identifying and maintaining trace dependencies leads to additional effort that can get prohibitively expensive with increasing number of requirements and increasing tracing precision. In practice, tracing is typically not an explicit systematic process, but occurs rather ad hoc with considerable hidden tracing-related quality costs.

Methods, tools and approaches of requirements tracing reported in literature [77][112] provide technical models about how to store identified traces. Most tools aim to automate requirements tracing, but tracing automation is still complex and error prone. Furthermore, automation alone cannot really reduce efforts of requirements tracing. Thus, requirements tracing may seem too costly for routine use in practice. A major reason is that existing approaches make no difference between requirements that are very valuable to trace and requirements that are much less valuable. Tracing value depends on parameters like stakeholder importance, risk or volatility of the requirement, and the necessary tracing costs. Thus, there is the need for requirements tracing approaches that take these parameters into consideration, such as value-based requirements tracing (VBRT).

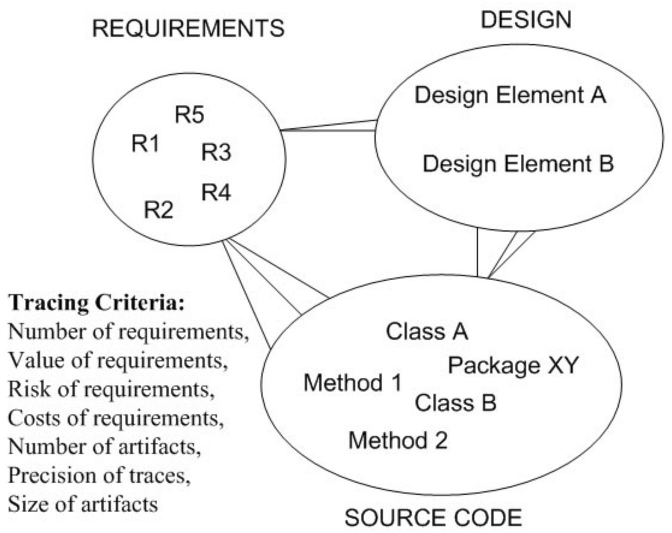


Figure 29 Requirements tracing overview

Systematic full tracing, where every requirement is traced with the same precision independent of its value, provides benefits in saving time in implementing error reports or change requests after the project has been finished. In this case the costs for new-coming maintenance personnel to re-discover knowledge about interdependencies in the system would usually be much higher than for identifying and storing trace dependencies during development with the original developers present. It seems easier to identify traces during development than later after project completion when a change request occurs.

Capturing all requirements traces can get complex and expensive very fast, e.g., imagine a software development project with only 18 requirements but more than 4000 traces to store and maintain. In comparison to full tracing, VBRT promises in a typical project significant reduction of tracing costs without losing its benefits.

The VBRT approach provides a technical model and an economic model for requirements tracing, depending on criteria like number of requirements, value of requirements, risk of requirements, number of artifacts, number of traces, precision of traces, size of artifacts, cost/effort of trace identification and maintenance, and value of traces. Figure 29 illustrates that traces can have different levels of precision, e.g., traces in code at method, class, or package level. Traces exist between all kinds of artifacts, e.g. design, code, test cases. Thus, VBRT can help to perform cost-efficient requirements tracing within given budget limitations.

Project stakeholders like customers, project managers, and quality managers do not put equal value on each requirement. Value-based software engineering approaches such as release planning [118] relate value differences to project decisions and practice. Similar to the requirements themselves requirement traces are not equally important. However, existing approaches treat each trace the same way and do not consider these value differences. With limited resources the project team has to decide capturing which traces seems most worthwhile.

The VBRT approach consists of 5 steps: requirements definition, requirements prioritization, requirements packaging, requirements linking, and evaluation. VBRT reduces tracing efforts by prioritizing requirements. Requirements prioritization uses the input parameters stakeholder value, requirements risk/volatility, and tracing costs to decide which requirements are valuable enough to trace and which are not.

A main contribution of this paper is a case study where I applied VBRT to a real-life project and report initial case study results, e.g., tracing-related costs. The focus of this work was on traces between requirements and other artifacts, especially code pieces (vertical traceability), while horizontal traceability is part of further work.

In this work I evaluated the VBRT approach in a real-life project setting and compared costs and benefits of VBRT to ad hoc tracing and full tracing. The case study results suggest that VBRT can be an attractive tracing alternative for typical software development projects, because it provides "traditional" benefits of tracing and minimizes tracing efforts at the same time.

5.1.1 Value-based Requirements Tracing

The goal of the value-based requirements tracing process is to identify traces based on prioritized requirements and thus to identify which traces are more important and valuable than others. The following subsections provide a VBRT process overview, simple cost-benefit model, and research question.

VBRT Process Overview

Figure 30 depicts process activities, actors, and deliverables of VBRT. In an iterative life cycle the VBRT process represents one cycle of developing and refining the value-based traceability system.

The VBRT process consists of five distinct steps: (1) requirements definition, (2) requirements prioritization, (3) packaging of requirements, (4) linking of artifacts, and (5) evaluation.

During *(1) requirement definition* the project manager or requirements engineer analyzes the software requirements specification and identifies atomic requirements. The requirements engineer then assigns a unique identifier to every requirement. The result is a list of requirements and their IDs.

During *(2) requirements prioritization* all stakeholders assess the requirements and estimate the value, risk, and effort of each requirement. The result of this step is an ordered list of requirements where the requirements are ranked on three priority levels [98].

(3) Requirements packaging is an optional process step that allows a group of architects to identify clusters of requirements. These clusters are needed to develop and refine architecture from a given set of requirements.

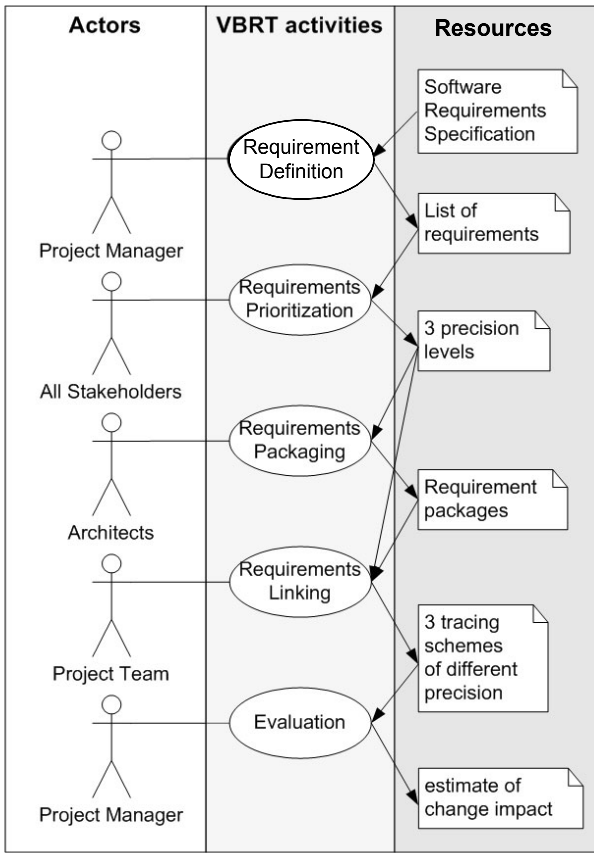


Figure 30 Value-Based Requirements Tracing Process Overview

During *(4) requirements linking,* the project team establishes traceability links between requirements and artifacts. Important requirements are traced in more detail than less important requirements. Therefore, I use 3 levels of tracing intensity. The result of this step is an overall traceability plan.

During (5) *Evaluation* the project manager can uses traces for certain purposes, e.g., to estimate the impact of change for certain requirements.

Cost-benefit Model for VBRT

Tracing techniques typically provide only technical support to perform requirements tracing but do not take value and cost considerations into account.

Complete tracing needs often prohibitive effort and duration in a software project. The question arises, how much effort for tracing is appropriate to provide significant savings during usage of traces in a project. I want to optimize the cost-benefit of tracing and trace analyses. I assume that value-based requirements tracing can help to find a subset of traces that saves proportionally more cost than it loses benefit.

The costs and benefits of requirements tracing depend on the following parameters.

Project context:

- *Number of artifacts* to be traced; the higher the number of artifacts, the higher is the effort to create traces between them. It depends on the trace applications (e.g., requirements conflicts identification, change impact analysis, consistency checking, and verification) which artifacts should be considered for tracing.
- *Number of requirements* in a software development project; Due to n^2 complexity of requirements tracing (potential traces between all n artifacts), the effort explodes with increasing number of requirements. That is one of the main problems of requirements traceability. VBRT is an approach to get a grip on the tracing effort problem.
- *Value of requirements* that is the importance of each requirement to the stakeholders (e.g., on a three-point-scale); I suppose that high-value requirements need more detailed tracing than low-value requirements, because they represent the core functionality of the system and trace information of the latter is more important than trace information of low-value requirements.
- *Risk of requirements*, that is the volatility of each requirement (e.g., on a three-point-scale); It seems to be worthwhile to trace risky/volatile requirements in more detail, because during trace applications like change impact analysis, these traces are needed more frequently than traces to stable requirements.

Tailoring parameters:

- *Number of traces*; the higher the number of requirements and artifacts to be traced, the higher is the number of potential traces to identify and maintain;
- *Precision of traces*, e.g., traces between requirements and code could be at lines of code level, method level, class level, or package level [35][39];
- *Complexity/Size* of trace objects, e.g., if a code class is extremely big, then tracing at method level would provide a higher value than tracing at class level. If the class is very small and contains only one method, then tracing at class level provides nearly the same value as tracing at method level.

Cost and benefit:

- *Cost/Effort for tracing*, e.g., more precise traces are more expensive to identify than less precise ones. Reducing the number of traces, e.g., by omitting tracing of less important requirements, also has an effect on costs and efforts.

- *Value of traces* in context of a specific application, e.g., change impact analysis, identification of requirements conflicts, etc. For example, in context of change impact analysis, using traces reduces costs and time for locating code pieces to be changed.

I compare three tracing alternatives in this paper. The first alternative is *ad hoc tracing*. The project team does not create and maintain any kind of traces during development, but searches documentation for relationships when needed. This variant has hidden efforts for search and rework risk.

The second alternative is *full tracing*. The project team does not make differences between requirements and traces each requirement with the same effort and precision. It makes a difference in this variant whether the project team identifies traces during development or after the project (ex post). The latter approach seems to be considerably more expensive than the first, as the project team has often to re-discover system details.

As described above, full tracing provides certain benefits in comparison with ad hoc tracing, but there is still a potential for improvement or optimization, e.g., full tracing wastes efforts for tracing requirements that do not really need to be traced with that level of precision. Thus, the criterion for optimization is which requirements should be traced at which level of precision.

VBRT addresses this issue by providing a requirements prioritization step where requirements are assigned to one of three precision levels. For example, a ratio of requirements per precision level of 10%:30%:70% or 20%:40%:40% would provide a considerable effort reduction. Thus, VBRT tailors tracing efforts down to manageable size without losing too much of the benefits of full tracing.

The scope and prioritization of requirements is important as not all requirements do have similar value and trace sets are usually not complete due to effort constraints and duration of trace creation in the software development process. Therefore, the question arises which traces are most worthwhile to create and maintain in a software project.

One aspect is the value of a trace set for the stakeholders; another is the risk of change volatility of a trace, and finally the cost of creating and maintaining traces. I performed a requirements prioritization step in our case study based on the prioritization approach by Ngo-The and Ruhe [98] in order to identify most important requirements, medium important requirements and less important requirements.

Research Question

In context with the VBRT process, this work deals with the following research question:

- RQ: To what extent can VBRT reduce requirements tracing efforts (economy of requirements tracing)?

I assume that VBRT reduces tracing efforts by omitting identification of unimportant traces through requirements prioritization. I measure the tracing effort in person hours to evaluate this research question.

I assume that traces differ in their value depending on requirements' value, costs, and risks (volatility). I evaluate this question by analyzing the results of the prioritization step, where requirements are assigned to precision levels. The most valuable, most costly, and most risky (volatile) requirements should be traced on the highest precision level.

To gather data to answer this research question, I performed a case study at Siemens Austria. Focus of the case study was to apply the VBRT process in a small project that allows comparing full tracing and VBRT effort and discussion of empirical data with development experts. The case study should be a basis for extrapolation of tracing and cost-benefit parameters to a typical larger project.

5.1.2 Case Study Application of VBRT

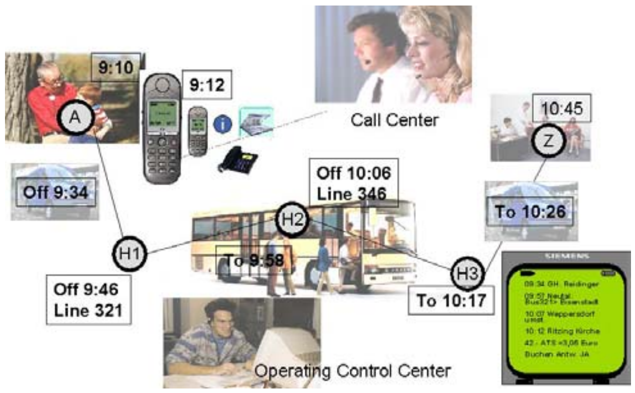


Figure 31 Communication example from case study project

The case study project “public transport on demand” is about an improved and more efficient public transportation system in rural areas supported with modern information technologies. The challenge is to stop further deterioration of public transportation access in rural areas with a new traffic model. The basic element in the system is a public transportation service provider centre (PTSPC). The passenger can ask the PTSPC via SMS, Internet or Call Center for transportation on a route within the service area. The passenger has to provide input parameters like starting point, destination, arrival or departure time, maximal amount of transfers, and maximal acceptable travel time. If the location of the start or the destination has no scheduled stop within walking distance, the system will arrange a feeder service to or from the stop. Passengers can ask the PTSPC for route information and prices; they can also directly buy their tickets. The PTSPC is thus able to calculate the best possible route and the price of the requested trip. Figure 31 illustrates the target traffic communication model: A customer orders a route with his handy via call center, the PTSPC calculates a route consisting of transport mode options, stops, and pickup times. Finally, the customer receives a SMS with the route information.

The PTSPC ensures that all orders are executed properly. The PTSPC also arranges for a follow-up acknowledgement between the feeder system and the buses. All vehicles of the feeder service and all the buses are equipped with location and communication devices. Therefore, the PTSPC knows the locations of all the vehicles in the system and can communicate with their drivers. The PTSPC operator is thus able to notify the drivers and arrange for appropriate actions to be taken in the case of major unexpected deviations from schedule.

The type and size of the project was suitable for us to apply the VBRT approach in a software development project with realistic yet manageable trace options. The project consisted of 46 requirements, which seemed to be the right magnitude to evaluate the VBRT approach, because the number of requirements was neither too high nor too little. The following artifacts existed when I started the case study:

- *Software requirements specification:* The specification contained the description of functional requirements. Non-functional requirements, e.g., quality, performance, reliability, were not described and therefore excluded from our case study.
- *Architecture description and high-level design:* These artifacts described the building blocks of the desired system.
- *Prototype:* The prototype was a partial implementation of the requirements in the software requirements specification.

The following subsections describe the VBRT process steps in context of the case study.

Requirements Definition

The prerequisite for requirements definition is a software requirements specification. This software requirements specification is written in plain text and most of the functional requirements are modeled as use cases. The software requirements specification contains 46 functional requirements. The main task of this process step for the investigator was to review the software requirements specification and to extract each use case title into an excel list.

Results of requirements definition

One person needed approximately 3.5 hours to generate the *requirements list* from a textual software requirement specification.

Requirements Prioritization

When I performed the case study the project team consisted of three project members: the project manager, the quality manager, and one additional project member. These three persons performed the prioritization step. The project manager had to assess the value, risk, and effort of each requirement. All other project members had to assess the value only (stakeholder value proposition). Table 18 illustrates a part of the project manager's prioritization sheet and Table 19 depicts a part of the standard prioritization sheet for all other project members. In order to support the understanding of each requirement, the working sheets contained short descriptions of every requirement and every requirement description contained a link to the relevant chapter of the software requirements specification. The project manager assessed value, risk, and costs of each requirement, whereas the rest of the project team assessed just the value of each requirement (on a scale with high importance +, medium importance 0, and low importance -).

Table 18 Prioritization sheet for the project manager

Requirements	**Value**	**Risk**	**Effort**
Req1. Registration, Login, Logout	-	-	0
Req2. Change user profile via internet	0	0	0
Req3. Order a route via internet	+	+	+
Req4. Pre-configure a route	+	0	0
Req5. Order a route via SMS	+	-	0
Req6. Order a route via call center	+	+	+

Table 19 Prioritization sheet for all other project members

Requirements	**Value**
Req1. Registration, Login, Logout	+
Req2. Change user profile via internet	0
Req3. Order a route via internet	+
Req4. Pre-configure a route	+
Req5. Order a route via SMS	+
Req6. Order a route via call center	+

Based on the three project members' assessments, I calculated a general result table. I counted the number of +, 0 and - from the value assessment. Based on these counts, I calculated the overall stakeholder value classification, SV (see Table 20).

The classifications of risk R and effort E, performed by the project manager, resulted in the classification RE, reflecting the overall risk/effort situation for every requirement, ranging from '--' very low to '++' very high. Both the stakeholder value classification, SV, and the risk/effort situation, RE, were input to determine a priority level L, ranging from 1 – high priority – to 3 – low priority (see Table 20). The prioritization approach is described in [98] in detail.

Results of requirements prioritization

The project manager assessed value, risk, and effort for each requirement, whereas other project members assessed only the value. The duration of the prioritization step took per person between 40 and 60 minutes. The assessing project members did not have to cooperate but performed their assessment individually. Table 20 depicts the overall assessment of the requirements list.

The columns value +, 0, and - contain the number of votes, e.g., 2 project members voted for Req1 to be important (+), and one project member voted for Req1 to be unimportant (-). The columns R and E contain the project manager's assessment of risk and efforts for each requirement. The column RE contains the combination of R and E; column S contains the combination of the value assessments. Finally, column L contains the assignment of each requirement to one of three priority levels, e.g., Req1 is assigned to level 2 (medium importance) and Req3 is assigned to level 1 (high importance). The prioritization step is described in Ruhe [98] in detail.

The VBRT prioritization step has the following characteristics:

Prioritization is short. In the case study the average duration for the stakeholders' prioritization was 50 minutes. The number of requirements in each priority level seems to be suitable. The distribution of requirements to the three priority levels was approximately 1:6:2, that means out of 9 requirements 1 requirement was priority level 1, 6 requirements were priority level 2, and 2 requirements were priority level 3.

Table 20 Stakeholder requirements prioritization results

List of requirements	Value			R	E	RE	SV	L
	+	0	-					
Req1. Registration, Login, Logout	2	0	1	-	0	-	+	2
Req2. Change user profile via internet	0	3	0	0	0	0	0	2
Req3. Order a route via internet	2	1	0	+	+	++	+	1
Req4. Pre-configure a route	2	0	1	0	0	0	+	2
Req5. Order a route via SMS	2	1	0	-	0	-	+	2
Req6. Order a route via call center	3	0	0	+	+	++	+	1
Req7. Change the user profile via call center	1	2	0	-	-	--	0	3
Req8. Administration of orders via internet	3	0	0	+	+	++	+	1
Req9. The driver may see order details of his orders	2	1	0	0	0	0	+	2
Req10. Data transfer between taxis and the central dispatcher	1	2	0	-	+	0	0	2

Requirements Packaging

After the project team classified each requirement on one priority level, the next optional step is about generating an architecture proposal by means of an intermediate model. This requirements packaging step is not described in detail because it is optional and not directly within the focus of this paper.

Requirements Linking

Requirements on priority level 1 are traced in more detail than requirements on priority levels 2 and 3. This graduation of intensity reduces the overhead for tracing unimportant requirements and provides only really necessary information about dependencies between requirements and artifacts.

In this case study the investigator performed the linking step. The investigator started from the user interface of a prototype and tested one requirement after the other. At the same time, he investigated which code pieces were invoked (methods at priority level 1, classes at priority level 2, and packages at priority level 3). He did this investigation by inspecting the code manually. So, the investigator did not perform the linking step concurrently to developing but 'ex post'.

Level 1 linking is the most expensive linking, because the investigator had to link each requirement to every method invoked during its "performance". The investigator developed a traceability matrix containing the requirements as columns and code methods as rows. If, for example, method A participates in the implementation of requirement B, then the cell where the row of method A crosses the column of requirement B contains an 'X'. Each method identifier contains the package name, the class name and the method name, each separated by a period. After the investigator finished level 1 linking, he continued and did the same for precision levels 2 (class level) and 3 (package level).

Results of requirements linking

The effort to create trace links into code at method level is rather high (ca. 45 min per requirement), but, on the other hand, it seems to provide the most useful information with respect to traceability.

Linking into code at class level does not need very much effort (ca. 10 min per requirement). The usefulness of this information depends on class size. For small to medium classes, the level of detail of this information is sufficient to locate the code relevant to a certain requirement. For large classes, e.g., implementing dozens of methods, level 2 linking gives only little support.

Linking into code at package level is done very quickly but does not provide very useful information. Linking at package level therefore is sufficient only for unimportant requirements.

In the case study, requirement linking was performed ex post; therefore it was harder to get an insight into the system. If the linking is done during the project, efforts should be reduced.

The third type of tracing concerns requirements on the lowest precision level (package level). The case study pointed out that tracing at package level can be generally left out, because the resulting traceability matrix provides only very coarse information.

5.1.3 General Case Study Results

As mentioned above, tracing into code at method level provides more precise information than tracing at class or package level. Unfortunately, the effort for tracing into detailed code is usually very high. The case study pointed out that focusing on the most important requirements reduces efforts in comparison with tracing all requirements at method level in the case study. The total effort for VBRT requirements linking was 770 minutes. That means, it took some 13 hours to establish a VBRT requirements traceability system for a software project with 46 requirements. In comparison, tracing all requirements at method level would have taken 2070 minutes (some 35 hours). Thus, VBRT used around 35% of the effort to establish a full requirements traceability system.

Another interesting point is the usability of VBRT in comparison with ad hoc tracing. The project manager of our case study recognized requirements traceability as additional, time consuming, and expensive effort. In the case study the investigator identified traces ex-post and had problems to get into implementation details, whereas the developers during implementation do not have this problem. So, the case study suggests that capturing traceability information in early phases of the software development lifecycle is much easier than capturing traceability information in later phases. I want to evaluate this hypothesis in further work.

Another issue is the level of detail or precision of tracing. Due to effort and budget constraints it is often impossible to trace each requirement at highest level of detail. Value-based approaches allow tailoring efforts according to requirements' priority. In context with requirements traceability, I interpreted this as to use trace types of variable precision. For example, I used three different trace types to trace the requirements into code, namely method traces, class traces, and package traces. The first trace type allows tracing requirements into code at method level, the second trace type allows tracing requirements into code at class level, and the third trace type allows tracing requirements into code at package level. This reflects a level of precision and also effort necessary to create these traces, e.g., tracing into methods is more expensive than tracing into classes.

It is common knowledge that generally tracing requirements into code at method level provides more detailed and usable information than tracing into class and package level. This is especially true for code that consists of very long source code classes, because the latter often contain methods implementing different parts of functionality. So the information "requirement A is implemented by methods 1 and 2" is more useful than the information "requirement A is implemented in class X", because there could possibly be many more methods in class X that do not relate to requirement A.

This higher quality of information has its price in effort necessary to create these traces. For instance, the case study presented in this paper pointed out that tracing a requirement into code at method level needed one person for 45 minutes on average to create this trace. Tracing the same requirement into code at class level took only 10 minutes. Tracing the same requirement into code at package level took only 2 minutes, but these traces have very little benefit, because the resulting traceability information is much too coarse, whereas tracing at class level turned out to be sufficient when source code classes are short and clear.

Another question was how the "risk" of a requirement has an impact on the detail of tracing. Most risky requirements are prone to changes and also need many cycles of adjustment during the process. Thus, it is important to understand the impact of requirement changes on system design and other development artifacts with high precision. This implies that tracing of most risky requirements with high precision has a high benefit, because these traces are needed very often, e.g., during change impact analysis. Furthermore, tracing of risky requirements must be both cheap and fast to allow unobtrusive trace analyzes during software development. Tracing risky requirements also supports the design principle of dividing volatile and less volatile requirements in the design structure.

5.1.4 Discussion

The high effort of requirements tracing seems to be a main reason why project teams do not use requirements tracing in practice. Most automation approaches alone do not suffice to tailor down tracing efforts to a manageable size, because they do not reduce complexity of tracing, e.g., the number of traces.

The case study results pointed out that the VBRT approach allows reducing tracing efforts without losing significant requirements tracing benefits. In comparison to full tracing, VBRT took only 35% effort. Thus, VBRT is a good step towards solving requirements tracing problems like high efforts and high complexity.

Furthermore the case study showed that identification of traces early in the project lifecycle is easier than in later phases. In later phases, e.g., in the operation phase, the rework to get into program details again is considerably higher. This is generally a good argument for requirements tracing, because capturing traces during development is economically much more worthwhile than on the occasion of change requests etc, when the actors do not know implementation details and spend lots of effort to understand the latter.

The prioritization step of the VBRT approach is a suitable means to identify which requirements are more valuable to trace than others. This prioritization is based on requirement parameters like value, risk, and costs and results in reduced efforts, because less important requirements are traced with less efforts and more valuable requirements are traced with more detail. Of course, the prioritization of requirements by the stakeholders is subjective, because it is based on the stakeholder value proposition.

In the case study there were approximately 10% of all requirements in precision/priority level 1, 60% in level 2, and 30% in level 3.

The case study pointed out that all requirements that the stakeholder assessed as high risks (on a scale ranging from high risk, medium risk, to low risk) were assigned to the highest priority level. Thus, I traced them with highest precision. The high risk of these requirements is synonymous with the high volatility of these requirements. That means that requirements with a high probability of change are very valuable to trace.

Based on the case study results, a comparable larger software development project is likely to have the characteristics illustrated in Table 21 that depicts strengths and weaknesses of ad hoc tracing, full tracing, and value-based requirements tracing. In the right-most column the optimistic case assumes very little need for extra traces, while the pessimistic case assumes a need for extensive traces to support project activities on the critical path. The overall cost comes from pro-active trace creation and reactive work on tracing when actually needed. I assume the overall tracing-related effort for full tracing in larger projects to be on average approximately 5% of the total project costs as part of quality assurance activities, such as testing or inspection, where traces are a perquisite for a sound quality assurance plan. Ad hoc tracing is likely to cause on average similar but hidden costs, while the cost variation in projects may be very high.

At first sight, *ad hoc tracing* seems to be the cheapest alternative, as there are no costs for trace identification and maintenance. In projects where requirements changes are very likely this alternative might become very costly, because the project team or maintenance personnel have to do “tracing” ad hoc. The later these change requests happen, the more costly tracing gets during development. Further, tracing efforts on activities that are on the critical path for project or maintenance task completion will effectively delay the overall finish. Omitting tracing at that point incurs a high risk of lower-quality solutions and/or erosion of system design [12].

Table 21 Comparison of tracing alternatives

	Proactive effort to identify and maintain traces	Additional effort and delay in case of a change request (reactive)	Overall tracing cost in % of total project costs (optimistic to worst case)
Ad hoc Tracing	Low (0)	Extremely high	0% to 20%
Full Tracing	High	Low	5% to 15%
VBRT	Medium	Low	2% to 7%

The second alternative is *full tracing*. Its effort for trace identification and maintenance is high, because every requirement is traced with the same precision, although many requirements do not need to be traced. Thus, effort is wasted with this variant on many less important requirements, which makes it rather unattractive for practitioners. The general benefit of requirements tracing is the lower delay and lower additional effort in case of a change request.

The third alternative is VBRT. The effort to identify and maintain traces is less than half of full traces and the additional effort in case of change requests is low. Thus, VBRT provides similar value as full tracing, but is much cheaper. Based on our assumptions for a typical project and case study results, the overall tracing effort in a large project with VBRT is likely to be fewer than 3% of the total project costs. This reduction makes tracing advisable in practice, especially after making the hidden costs of ad hoc tracing visible.

Tracing effort depends in practice mostly on the parameters: number of traces, level of detail of traces, change rate of traces at the occasions when tracing is done, e.g., weekly to have a current picture; or, at milestones when artifacts reach a stable state. The number of traces depends on the system artifacts and their size and complexity; the change rate on project context. However, the project manager can control the level of detail of traces and the occasions when tracing gets conducted.

The accurate estimation of tracing effort in general software engineering projects is very difficult as these efforts are often hidden as part of engineering and quality assurance tasks. However, based on a an analysis of the tasks of quality assurance and the amount of effective tracing work involved, I can as initial estimate assume that tracing will consume around a third of quality assurance effort in projects with good quality assurance support. Based on this assumption and typical quality assurance efforts in software development and maintenance projects, the effort estimates in Figure 4 for trace identification during the project and as reaction to change requests seem reasonable. The case study showed that prioritization of requirements by the stakeholders is an effective approach that can lead to a reduction of tracing effort in practice between 30% and 70%. However, the overall savings in a project context depend, of course, on many factors, e.g., how well the system documentation is organized and the overall complexity of the system artifacts. VBRT aims to reduce the inevitable effort for tracing to a level that makes effective tracing more attractive to practitioners.

Validity of results: The purpose of the case study presented in this paper was to investigate the impact of full tracing vs. VBRT on effort and benefits. Thus the case study size was chosen to allow conducting both full tracing and VBRT in a reasonable amount of time. However, the case study project setting is typical in the company and allows reasonable insight into the feasibility of the VBRT process in this environment. I see the empirical investigation as an initial study that supports planning further empirical studies with larger projects. As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings.

5.1.5 Conclusion and Further Work

In software development projects there are interdependencies between all kinds of artifacts, e.g. requirements, design, source code, test cases. Requirements tracing is the ability to follow the life of a requirement in a forward and backward direction [51] and helps project managers and project teams to make this interdependencies transparent. Capturing these interdependencies (traces) explicitly brings benefits for identification of requirements conflicts, change impact analysis, release planning etc., but the high complexity and necessary effort of tracing makes requirements tracing too costly for use in practice. Existing methods, tools and approaches of requirements tracing in literature provide only technical models about how to store identified traces and do not take this economic issue into consideration.

In this work I evaluated the VBRT approach in a real-life project setting and compared costs and benefits of VBRT with ad hoc tracing and full tracing. For the purpose of evaluation, the project team performed the VBRT process steps. I then analyzed the results and compared it with ad hoc tracing and full tracing. Main results of the case study were: (a) VBRT took around 35% effort compared to full tracing; (b) more risky requirements need more detailed tracing. The case study results illustrate that VBRT is an attractive tracing alternative for typical software development projects in comparison with ad hoc tracing and full tracing, because it provides "traditional" benefits of tracing and thereby minimizes tracing efforts.

For a more general evaluation of VBRT and to evaluate the cost difference of VBRT and full tracing in the face of changing requirements I plan multiple case studies with a systematic range of projects. I will address the question which level of detail is optimal to trace requirements into code, e.g. class or method level. Software engineering standards demand requirements traceability but do not state the required level of detail. Further case studies will explore the cost-quality trade-off of tracing at different levels of detail. Automation approaches for requirements tracing are also a future topic in context of VBRT. I want to use the trace analyzer tool [35] to explore the cost-quality trade-offs between automated tracing at method level and class level.

Another focus will lie on improvement of requirements prioritization in order to optimize the value of VBRT. There are many more relevant requirement attributes than value, risk, and effort that are interesting in context with prioritizing requirements for requirements tracing, e.g., architectural relevance, stability. Another open question is how VBRT can support horizontal traceability, because this paper focused on vertical traceability.

A third focus will lie on developing automated support assisting engineers in exploring and using the automatically derived trace dependencies. One idea is to integrate requirements traceability approaches into existing development environments, so that the developer can implement code and store traceability information simultaneously.

Requirements tracing is important to keep track of the interdependencies between requirements and other artifacts and to support project teams and software maintenance personnel in several tasks, e.g. change impact analysis, requirements conflict identification, consistency checking. VBRT is a promising approach to alleviate the problem of high effort of requirements tracing in a practical and comprehensible way.

5.2 A Value-based Approach for Understanding Cost-Benefit Trade-offs During Automated Software Traceability

Establishing and maintaining trace links places a big burden on software engineers. There are tools available that provide the infrastructure for managing trace links (e.g., case tools, requirements management tools).

However, these tools do not free the engineers from identifying links or from ensuring their validity over time. Traceability of any kind is therefore hardly adopted in industry, mainly due to cost and complexity issues [5].

Yet, the generation of trace links is increasingly mandated through standards and industry is moving to adopt these standards and even impose them on subcontractors.

There is thus a growing need to overcome the traceability problem and researchers have been developing approaches for generating trace links to help the engineers [5][22][35]. These approaches bring some relief but they rely on the quality of the input. Imprecise and sloppy input generally results in lower-quality trace links, i.e., false trace links (false positives) or missing trace links (false negatives). In the quest to produce a perfect set of trace links (without false positives or false negatives) one tends to forget that there is a significant cost-quality trade-off involved, which affects the usage intensity of trace links in practice.

Automatic Trace Link Generation

As manual trace link definition tends to be costly and error prone, we use in our studies an automatic approach, supported by the Trace/Analyzer tool [35]. This approach uses software-artifacts-to-code mappings as input and generates trace links among software artifacts as output. In simple terms, the Trace/Analyzer approach generates a trace link if and only if two artifacts overlap in their common use of source code. Such overlap may be obscured in various ways (e.g., uncertainty, grouping, utility code), but the results are still usable in practice.

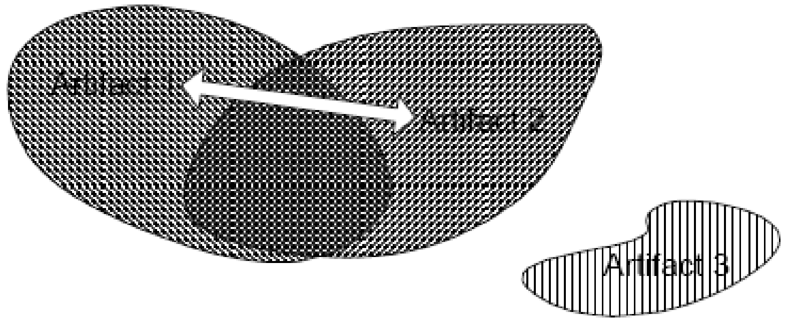


Figure 32 Trace/Analyzer generates a trace link between two artefacts with overlapping code[64]

However, the quality of the generated trace links is strongly affected by the level of detail of the source code. It is up to the engineers to define whether they map artifacts and code in terms of:

- requirements to Java package mappings,
- requirements to Java class mappings, or
- requirements to Java method mappings

Value-Based Software Engineering

We believe that value considerations are needed for planning software traceability in a sustainable way. Currently, the indirect contribution of trace links to product value often leads to a value-neutral and purely technical perception and underestimates the need to align the incentives of success-critical stakeholders to generate and use trace links to their optimal potential.

Some initial results have been reported that consider value aspects in requirements traceability [63]. However, these reports have not conducted a cost-benefit analysis to find out when and how intensive tracing in a specific context is worthwhile.

This paper proposes a value-based approach to trace generation and rework. The recently defined paradigm of value-based software engineering [10] brings a new view to the traceability generation and maintenance. A motivation for value-based software engineering is that "much of current software engineering practice and research is done in a value-neutral setting, in which every requirement, use case, object, and defect is treated as equally important" [10]. The premise of value-based software development is that not every software artifact is considered equally important. In the context of software traceability we thus hypothesize that not all trace links are equally important. Taking a value-based perspective can help save cost and by emphasizing investing effort on software artifacts with a perceived higher stakeholder value.

Research Questions: Cost-Benefit Trade-Off of Trace Link Generation Alternatives

The main issues of our research is to investigate the impact of trace link generation quality (effort/cost) on the quality of applications that analyze trace links, such as change impact analysis. Better understanding of this relationship allows better planning of trace generation that helps aligning the cost-benefit considerations of the involved stakeholders.

In this paper we discuss three key trade-off issues for planning the trace generation process: (a) the level of detail of traces among artifacts (package, class, method levels); (b) the value of the artifacts that are traced (high-value artifacts justify a higher level of tracing effort); and (c) the points in time of trace generation (early vs. late). We present cost-benefit considerations, empirical data, and argue for a pragmatic value-based planning approach.

The simple question we asked ourselves was whether increases in quality of trace links do justify their cost. While it is out of the scope of this paper to provide a generally valid answer, we can describe costbenefit implications of trace generation and later trace link rework. We also suggest value-based software engineering [10] as a possible solution to maximize the benefits of trace links in relation to their cost. The following three questions are discussed in more detail in the remainder of this paper:

1. Is it necessary for trace link analysis to have as input a perfect set of trace links? Are false positives and false negatives acceptable? What are the implications of errors in trace links?

2. Is it necessary to have every trace link on the same level of detail? Are trace links of different quality acceptable?

3. What happens if we discover during trace analysis that a previously computed trace links is of insufficient quality for the planned purpose?

5.2.1 Quality Issues of Trace Dependencies

The benefits of trace links are a direct function of their usefulness to applications/humans that consume them. Trace links are consumed by applications that analyze trace link relationships, such as requirements conflict analysis, consistency checking, and change impact analysis. Engineers use them for navigation purposes to quickly locate related modeling data. The benefits of trace links depend on: the project context, the contribution of the application to the project, and the quality of the traces as input to the application. The benefit thus has to be determined in context as precondition to optimize the investment in input. Trace links are either generated manually (by the engineers) or (semi-)automatically based on some initial input. Furthermore, trace link can be generated as soon as new artifacts are created or as late as possible (right before the analysis of trace links). These parameters may have significant impact on the cost and benefit of trace links in a project.

Manual trace generation is essentially an ad-hoc process with little systematic guidance. Thus, the quality of traces may be insufficient for their application. Also, the generation of trace links at application time (this is the time the links are needed by an application) may delay the application significantly at a sensitive point of time in the project, e.g., during final stages before acceptance by the customer. Furthermore, the original developers may no longer be available or important details of their work may have been lost leading to a much more expensive and error-prone identification of the trace links.

(Semi-) automated approaches typically require some input but are able to compute (some) trace links without additional intervention and at no/low additional cost. In case of the Trace/Analyzer approach, the input is given in form of software artifacts to source code mappings. This input has to be generated manually or through testing as was discussed in [35]. The advantage of the Trace/Analyzer approach is that the required input only rises linearly with the size of the software product although the number of trace links normally rise exponentially. As discussed above, the input to the Trace/Analyzer can be provided at arbitrary levels of detail – mappings between artifact and packages, classes, methods, or, even, lines of code. Irrespective of the level of detail, the input may contain errors (i.e., a wrong or missing mappings) negatively affecting the correctness of the trace links.

5.2.2 Key Decisions of Trace Planning

In this section we take a look at three key dimensions of trace generation that have an impact on trace planning and the cost-benefit of the involved stakeholder: (a) empirical data on the investment into trace links at different levels of detail; (b) trade-off models for the investment in links among artifacts of different levels of value; and (c) a cost-benefit trade-off example regarding investment at different points in time;

Empirical Studies on Level of Detail and Quality of Trace Link Generation

In recent empirical studies [63] we found only limited economic value in improving the level of detail of trace links beyond a certain level. We observed a diseconomy of scale: the quality of trace links did not rise linearly with the effort invested for tracing on a more detailed level, but additional investment had lower additional impact on the quality of generated trace links (depicted below in Figure 33 for the opensource ArgoUML modeling tool suite).

ArgoUML consists of 49 packages, 645 classes, and over 6000 methods. The requirements-to-classes-input was an order of magnitude more expensive to generate than the requirements-to-packages-input. In turn, the requirements-to-methods-input was another order of magnitude more expensive to generate. However, one would expect that an increasing level detail would define the code overlap in more exact terms and thus produce better quality trace links. We thus hypothesized that the level of granularity of the input directly affects the false quality of the generated links. Figure 33 confirms this hypothesis by depicting a decreasing rate of false positives (y-axis) with increasing level of detail (x-axis). However, the number of false positives was not reduced at the same rate.

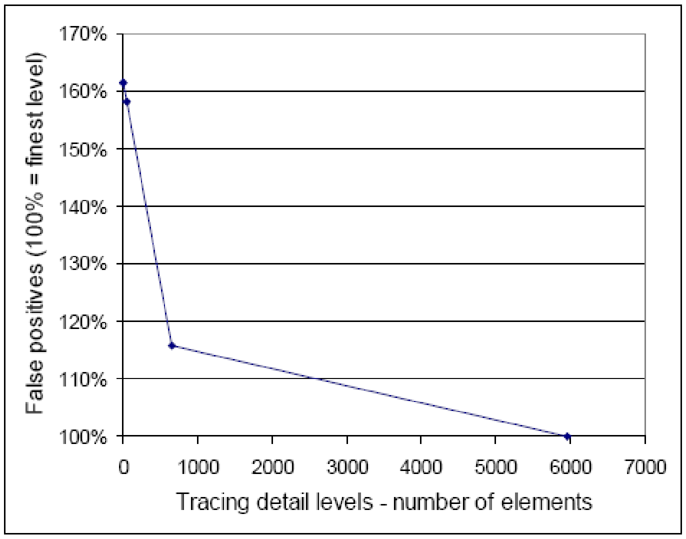


Figure 33 Marginally decreasing share of false trace links with increasing input level of detail[64]

A 10-fold effort increase by providing input in form of classes (more detail) instead of packages only resulted in a 42% reduction of (the known set of) false positives in the set of generated trace links among the input software artifacts. This is only a 2-fold increase in output quality for a 10-fold increase in input cost. Even worse, another 10-fold increase in input cost by providing the input in form of mappings to methods instead of classes only resulted in an additional 16% reduction of (the known set of) false positives. This is a rather low increase in quality for another 10-fold increase in cost (see incline of slopes in Figure 33).

We conducted similar experiments with two other case studies and observed a similar diseconomy of scale. It must be noted that these experiments only considered the initial cost of generating trace links with the Trace/Analyzer approach. There is also a cost for maintaining trace links over time. Thus we assess a lower boundary of the true cost of traceability.

Value of Artifacts and value of Trace Links

It is generally true that applications consuming traceability links can produce 100% correct results only if the input is 100% correct (in some cases not even then). Consumers of trace links are no exception. Since it is hard to produce 100% correct and complete trace links, it is clear that the application will suffer. We showed earlier that it is one order of magnitude cheaper to produce input for the Trace/Analyzer on the level of artifact-to-class mapping instead of the artifact-to-method mapping. This significant saving only results in a 16% quality reduction (false positives) of the resulting trace links.

However, such an across-the-board quality reduction is a value-neutral solution because it affects the quality of all trace links equally. While an engineer may be willing to sacrifice benefits to save cost, we believe that such a process must be guidable. A better solution would be to initially generate trace links of some minimal quality and then rework them later on if they are needed at a higher quality. This solution assumes that:

- Not every trace link is needed: generating and reworking trace links is wasteful in cases where they are not needed
- Some applications may only require a certain quality: generating and reworking trace links is wasteful if the quality improvements do not translate into benefits

Value-based software engineering places value on different software artifacts. For example, requirements can be classified as *critical*, *important*, or *nice to have*.

Even if the "nice to have" requirements are implemented, their correctness is not as important as in the case of "critical" requirements. We believe the Trace/Analyzer should be enhanced to consider such value information through the use of granularity:

- Low-value artifacts are mapped to the class level;
- High-value artifacts are mapped to the method level.

Table 22 artefact value and resulting trace link quality[64]

		Artifact 1	
		Low value	High value
Artifact 2	Low value	Low-detail trace link	Medium-detail trace link
	High value	Medium-detail trace link	Highest-detail trace link

In other words, a higher-value artifact, such as a requirement, is mapped on a finer level of detail than a lower-value artifact. Since the Trace/analyzer determines trace links based on overlaps, it will produce highest-quality trace links among high-value artifacts because of their finer-grained overlaps. Likewise, the quality of trace links among lower-value artifacts is lower because their overlaps are based on coarsergrained level of detail. Table 22 summarizes the four types of overlaps and their quality implications. The value classification thus directly translates to quality implications for trace links – and even more importantly, it also translates to their use in applications. For example, the requirements conflict analysis produces higher quality conflicts among higher quality requirements.

However, this solution still places equal value on all pieces of source code. That is, the artifact-tocode mapping is done equally for the entire code of an artifact. This is also unnecessary. Trace links are established on the basis of overlaps. If there is no overlap between two high-value artifacts, then there is no need to create artifact-to-code mappings entirely on a finer level of detail. Only the overlaps among high-value requirements need a finer-grained level of detail. This is because the quality of a trace link is a direct result of the *weakest level of granularity* of the involved artifacts.

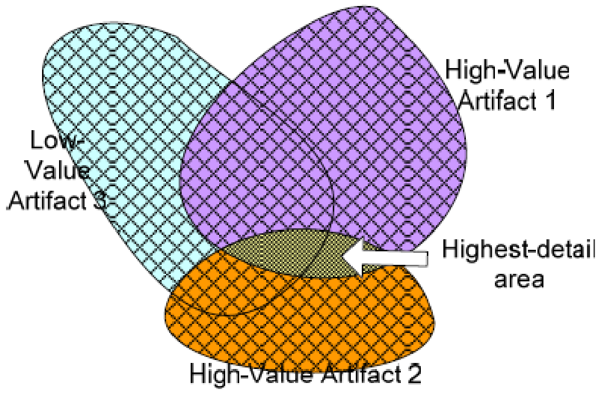


Figure 34 High level of detail only necessary for overlaps among high-value artifacts[64]

Figure 34 depicts this issue for three artifacts where artifacts 1 and 2 are of high value and artifact 3 is of low value. Areas of artifacts that do not overlap with other artifacts do not cause trace links. There is thus no benefit in investing effort into these areas. Areas where a high-value artifact overlaps with a low-value artefact only need to be considered on the level of granularity of the low-level artifact. There is no benefit in investing effort in increasing the granularity of one of the overlapping artifacts without doing the same for the other artifact. Only the overlapping areas of high-value artifacts should be of the same, finest level of detail.

But how do we know about the overlapping areas before we have gathered and analyzed the input of the Trace/Analyzer? This can be done by initially requiring all input in form of the coarsest-level of detail. The mapping of those overlaps can then be refined, if they belong to high-value artifacts.

Investment into Tracing at different Times in Development

The trace links are, by themselves, of little benefit. Their benefit is usually a direct result of their usefulness to support applications that require trace links as input. For example, we previously developed an approach to requirements conflict analysis that takes requirements, requirements trace links, and requirements classifications as input and computes potential requirements conflicts as a result. A false trace link (false positive) may result in a false conflict and a missing trace link (false negative) may result in a missing conflict. The quality of the trace links thus directly affects the quality of the requirements conflict analysis (an application).

If the quality of the generated trace links is low the quality of the results produced by applications consuming these links is also reduced. Thus, an important value consideration is how good is "good enough"? Is there a quality threshold trace links must meet for them to be useful for follow-on applications? Since low-quality traces are cheaper to produce than highquality traces, a follow-on question is whether there is a cost-benefit trade-off where the cost of producing higher-quality traces is higher than the benefits gained by their use? Recall that in case of the Trace/Analyzer approach, it takes an order of magnitude more input cost to marginally improve the quality of trace links. It is not obvious that this increase in input cost is justifiable in a project context.

Figure 35 depicts three options for possible costbenefit implications of trace generation, based on the assumption of less-than-perfect traces, i.e. daily tracing reality. Trace generation requires a certain input and produces some trace links as output. The cost of this input, typically directly related to effort (usually with some manual overhead), must be offset by the benefits gained from the results of the trace analysis applications.

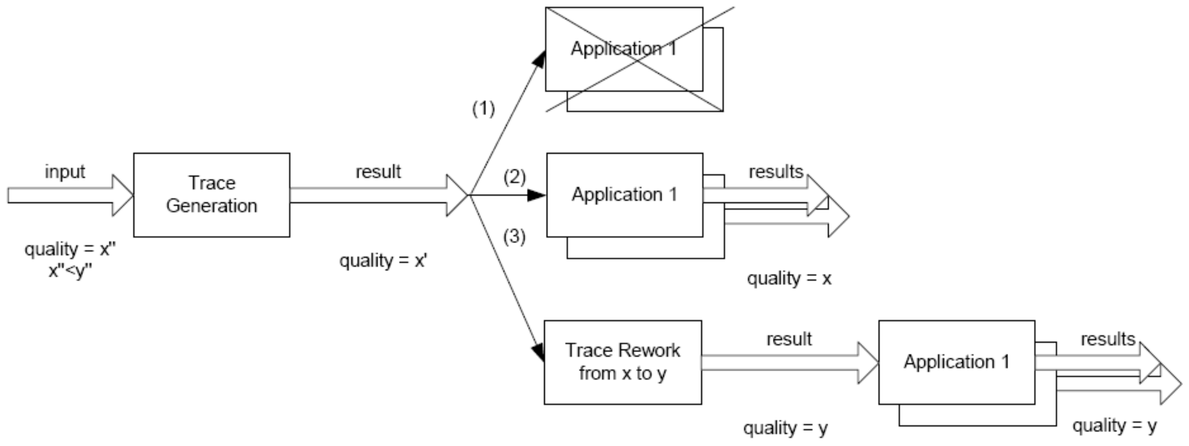


Figure 35 The cost-benefit implication of trace generation and trace rework[64]

If the quality of trace links is below the level of usefulness for some application, then the trace links serve no real purpose. Thus, the cost can never be recouped as there are no benefits. Trace generation in this situation is not economical (option 1).

If the quality of the trace links is above the usefulness threshold then the trace links are useful to applications and generate some benefit. The benefit is offset not only by the cost of the trace generation but also the cost of the application. However, an important consideration is that higher-quality trace links do not necessarily translate into more benefits. As we have seen in Figure 33, increasing the quality of trace links can come at a high price and its cost may never be recouped by the application. Too high-quality trace links may thus also not be economical (option 2).

If the trace generation produces trace links of insufficient quality then there is the option of a later trace rework (option 3). Trace rework improves the quality of trace links, but the cost of trace generation followed by later trace rework is likely to be higher than having done the initial trace generation to the desired quality because (1) knowledgeable engineers may have left the project or (2) they may not remember the solution details well enough. Trace rework is thus a way of improving the quality of trace links to make them useful for applications but at the expense of additional cost which reduces the cost-benefit ratio.

Impact of change over time on trace quality

The initial saved effort on trace generation is counteracted by loss of rework in case the less detailed traces are inadequate. The amount of rework depends on how much of the work not done has to be done and how much more difficult this is relative to generating a trace at development time by the original developers. However, trace links also degrade over time while the software product evolves. Consequently their application suffers. Therefore, the benefits of the application of trace links change with time even if the input stays the same (assuming that the software product is evolved during that time). Even (semi-)automated approaches are affected by this degradation. For example, every source code change potentially affects the mapping between the artifacts and source code and thus the input to the Trace/Analyzer approach may become increasingly incorrect over time. It follows that the trace links generated by the Trace/Analyzer approach decrease in quality over time. Trace rework is thus necessary even if the initial trace generation produced sufficient quality trace links. To minimize the cost of trace rework, it should be done at the same time the software product is changed to avoid delays during their application and to benefit from the fresh knowledge. Still, it is not obvious what changes to a software product cause changes to its trace links, which may keep engineers from keeping trace links current and thus loose the potential benefit.

In summary trace planning has to face difficult decisions:a low trace quality may be a cost saving measure initially but it may factually be counter productive because low-quality trace links may not be useful later and thus generate no benefit. A high traceability quality may be needlessly expensive and thus may also be counter productive. And, the cost of trace rework must be considered, especially if the trace links are generated early on while the software product evolves.

5.2.3 Conclusions and Further Work

As traceability is mandated by software standards, software engineers and managers need support to plan the generation of trace links: (a) the level of detail of trace links between artifacts and (b) the effort for trace generation at different times during development. In this paper we applied principles of value-based software engineering to traceability and raised the issue of the value of trace links and the level of effort investment into generating and maintaining/reworking trace links.

Based on an initial cost-benefit model we explored several options to guide the effort of trace generation with three parameters: (a) the level of detail of traces among artifacts (package, class, method levels); (b) the value of the artifacts that are traced (high-value artifacts justify a higher level of tracing effort); and (c) the points in time of trace generation (early vs. late).

While we could show the need for better understanding the cost and benefit of both trace generation and usage during trace analysis, we see some fundamental open issues that need further work and discussion at the workshop:

- How to determine the benefit of traceability in some tangible measure such as “saved engineering hours” that allows balancing these benefits with the investment into trace generation.
- How to describe the relationship between the quality of input traces to trace analysis application and the quality of the output of such analysis applications (compare Figure 34).

In our opinion the ability to answer these questions will largely determine whether an alignment of the stakeholder views on cost and benefits of tracing can be achieved, which in turn will determine the rate of adoption of tracing in practice.

5.3 Determining the Cost-Quality Trade-off for Automated Software Traceability

Approaches for establishing traceability between software artifacts such as user needs, requirements, architectural elements, or code play an important role in both software engineering research and practice. The topic has been researched for more than a decade [51]. Furthermore, numerous major software engineering standards such as the CMMI or ISO 15504 consider software traceability as a ‘best practice’ and mandate or strongly suggest the use of traceability techniques. In many branches of industry these standards are imposed on subcontractors. For example, several European car makers are demanding the fulfillment of a subset of ISO 15504 from all their subcontractors, who suddenly face the challenge of quickly and efficiently introducing traceability techniques within their organization.

Trace links support software engineers and quality assurance personnel during software development by helping them understand the many relationships among software artifacts. This is most worthwhile for large, complex, and long-living systems where there are many non-obvious relationships among artifacts. Trace analysis reveals relationships among a broader set of software artifacts such as user needs, requirements, architectural elements, or source code. For example, trace analysis can reveal which elements of a state chart or class diagram realize a requirement, or how the elements inside a state chart diagram relate to a class diagram, given that every state transition describes a distinct behavior and every class implements that behavior in form of a structure. While it might be easy to guess some of these trace dependencies, the semi-formal nature of many modeling languages (e.g., UML) and the informal nature of the requirements often make it hard to identify trace links completely.

Software traceability deals with (a) trace link generation, i.e., the manual and/or automated identification of trace links; and (b) trace link consumption, i.e., the use of trace links for conflict analysis [38], consistency checking, and change impact analysis. The benefit of these applications largely depends on the quality of the trace links provided by trace generation. For the trace analysis to be useful, one also must understand for which particular engineering tasks it is done and how incorrect trace links might affect the outcome. The ultimate goal is to tailor the trace analysis to produce the desired quality of trace links with the least amount of input effort.

Higher-quality trace links (i.e., fewer false positives/negatives) allow the users getting their tasks done faster and better. For example, higher-quality traces result in fewer false conflicts reported during requirements engineering, or a more precise definition of the impact of a change request during system evolution.

While high-quality trace links are a desirable goal, they are typically not economical to produce as identifying and validating trace links can be a big burden in a typical project context [113]. There are tools available that provide the infrastructure for managing trace links (e.g., case tools, requirements management tools). However, these tools do not free the engineers from identifying links and from ensuring their validity over time. Despite its benefits traceability is therefore hardly adopted in industry, mainly due to these cost and complexity issues [63].

To overcome these problems, researchers have been developing different automated or semi-automated approaches for trace generation [5]: For example, Antoniol *et al.* [5] discuss a technique for automatically recovering traceability links between object-oriented design models and code based on determining the similarity of paired elements from design and code. Spanoudakis *et al.* [125] have contributed a rule-based approach for automatically generating and maintaining traceability relations. A forward engineering approach is taken by Richardson and Green [115] in the area of program synthesis. Traceability relations are automatically derived between parts of a specification and parts of the synthesized program. These and other approaches bring some relief, but they strongly rely on the quality of the input of trace link generation: mainly, the level of detail, e.g., classes or methods as points of reference, and the level of incorrect or missing trace links.

In this paper we focus on trace generation, which deals with (1) identification of trace link candidates; and (2) validation of trace link candidates. The choice of techniques used for trace generation influences the amount and quality of traces, i.e., the number of correct trace links; incorrect trace links ("false positives"); and unreported relationships among artifacts ("false negatives"). The validation step aims at reducing the number of incorrect trace candidates. The aim is to achieve a level of quality that is sufficient for the applications consuming trace links. Of course, the desired level of quality varies with the project context and applications.

5.3.1 Quality Considerations in Trace Analysis

Trace Analyzer [40] supports generating trace link candidates based on a small "seed set" of trace links that can come from a human expert or from logs of test cases that link requirements (and other artifacts) to source code pieces. The engineer then chooses the desired level of detail for tracing source code: packages, classes, methods, or lines of code. The Trace Analyzer simplifies the finding of trace link candidates among any kind of software artifacts (e.g., requirements, design model elements, and code). It requires as input some initial input how software artifacts relate to some common representation of the system, usually the source code. This relationship is typically explored through testing, where engineers are expected to supply a set of test scenarios that match the software artifacts. During testing engineers log the lines of code, methods, or classes that are executed by these test scenarios. Since it must be known how test scenarios match software artifacts, one can infer the software-artifact-to-code mapping.

The existence of a trace link candidate between any two software artifacts is determined by their code overlaps. If two software artifacts do not share any code part (e.g., lines of code, methods, or classes) during execution (i.e., they execute in distinct parts of the system) then no trace link is assumed among them. Note that a trace link is not a data/control dependency, but simply describes some correlation between two software artifacts. If two artifacts do share code, then there may be a trace link, if the shared code is application-specific.

Trace link candidates are thus computed by the Trace Analyzer based on the degrees of code overlap among software artifacts. Testing is a validation form that can not guarantee completeness (i.e., test cases may be missing). This naturally affects trace generation and the Trace Analyzer thus provides an input language that lets the engineer express known uncertainties [39].

The Trace Analyzer leverages manual trace generation by adding new trace candidates based on initially available trace candidates, e.g., from test suites for a requirement that relate it to the "footprint" of executed code elements. In this paper we focus on the aspect of incorrect trace candidates ("false positives") and missing trace candidates ("false negatives") as these can incur substantially higher cost for trace candidate generation and validation.

While the trace analyzer tool greatly simplifies trace candidate generation it is not fool proof and the input affects the validity and value of the output. This profoundly influences the utility of applications that consume trace links such as consistency checking, change impact analysis, or trade-off analysis during later development iterations or software maintenance. Moreover, a dilemma is that the quality of trace analysis is not just a factor of the correctness of the input. Even correct input may yield wrong traces given that we have choices in the level of detail of how trace analysis is done during development.

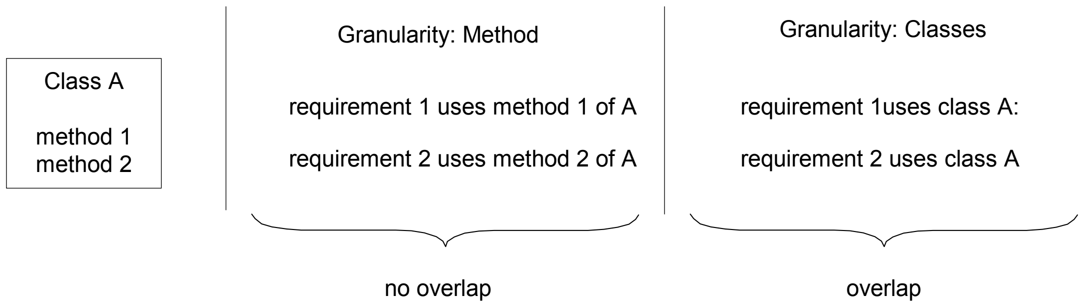

Class A
method 1
method 2
Granularity: Method
requirement 1 uses method 1 of A
requirement 2 uses method 2 of A
Granularity: Classes
requirement 1uses class A:
requirement 2 uses class A
no overlap
overlap

Figure 36. Code overlap example on method and class level[36]

Take, for example, in Figure 36 two requirements that are realized by the same class but by different methods. If we analyze traces on the method-level, then the two requirements do not overlap because they are executed in distinct methods. Yet, an analysis on class level will report an overlap because both requirements relate to the same class. The statement that both requirements execute inside the same class is correct, yet, may cause incorrect traces (i.e., false positives).

Standards such as the CMMI or ISO 15504 request the use of traceability techniques and also specify useful types of trace links. However, there are no explicit statements about the required level of detail of these links. A standard might be satisfied if software artifacts are mapped to classes instead of methods even though this introduces a significant number of flaws during trace analysis. Choosing the appropriate level of detail is a difficult problem: Method-level analysis allows more detailed results than class-level analysis; lines of code allow an even finer analysis. The execution of a particular if-statement inside a method may well point to the implications of a particular requirement. From the sole perspective of trace quality, the analysis on the level of lines of code is thus even more preferable.

While traceability on finer levels of detail promises a finer picture of the relationships among software artifacts, this is achieved at higher cost. A lower level of detail typically decreases trace quality because it results in a higher number of false positives (it is hard, however, to accurately estimate to what extent). It is thus vital for trace analysis to understand the cost-quality implication of different levels of detail. How much quality do we sacrifice by saving one-order of magnitude effort in input generation (which can be translated to engineering hours)? Is the sacrifice of quality acceptable?

5.3.2 Cost-Quality Trade-Off Analysis in Three Case Studies

In order to explore the cost-quality trade-offs just sketched, we conducted three case studies to investigate the effects of different levels of detail (i.e., method level, class level, and package level) for trace link generation on trace analyzer output quality (i.e., the level of false positives). The software systems we have chosen for our study are three differently-sized applications: the open-source ArgoUML modeling tool, a Siemens route-planning application, and a movie player. ArgoUML is an open-source software design tool supporting the Unified Modeling Language (UML). The entire source code and documentation for ArgoUML are available from *http://www.argouml.org*. The Siemens route-planning system, described in more detail in [63], supports efficient public transportation in rural areas with modern information technologies. The "Video on demand" package is essentially a movie player that can search for movies, select, and play them (more detailed information is available in).

Table 23 Case study systems and main context parameters[36]

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

Trade-off analysis of the level of detail vs. the level of quality. For each case study system we analyzed the impact of generating trace link candidates on the number of false positives. As baseline we took the level of false positives from trace links on the method level, the finest level of trace detail in our study. This analysis compares how a reduction of the level of detail (e.g., from method to class or from class to package) and the associated effort results in a higher level of false positives. For example, the largest of the three systems we evaluated, the ArgoUML system, has 49 packages, 645 classes, and 5,952 methods. Naturally, it is easier to map software artifacts to 645 Java classes than to 5,952 Java methods, and we can thus save at least one order of magnitude in effort by using classes instead of methods to determine the software-artifact-to-code mapping. We found that we can also save another order of magnitude, if we use packages instead of classes.

Strength analysis of trace link candidates. For the validation step after generating trace link candidates it is helpful to identify the candidates that are most likely false positives. Thus, we measured the "strength" of each trace link candidate. We expected to find a positive correlation between the number of false positives and the strength of trace link candidates.

The *"strength" of a trace link candidate* is defined as the ratio of the number of code elements (e.g., methods) that implement a requirement and the number of code elements that two requirements share as part of their implementation. For example, if requirement *X* is implemented in 10 methods, requirement *Y* is implemented in 5 methods, and 2 methods implement a part of *X* as well as a part of *Y*. Then, the trace strength for *X* is 2/10 = 20% and for *Y* 2/5 = 40%. Stronger trace links are less likely to be false positives. Our analysis once more underlines the diminishing return in investment with more detailed levels of input. Trace analysis on method level is in our study contexts around 10 times more expensive but produces almost no additional strong traces. These results are confirmed with the route-planning system data and the movie player. We found that during the validation of trace link candidates it is most worthwhile to concentrate on low-strength trace candidates. These trace candidates are harder to decide automatically and need human judgment. High-strength traces and traces with zero strength are very likely to be correct and we found in our studies little extra value in validating them manually.

5.3.3 Validation Support: Thresholds to Filter Error-Prone Trace Candidates

The data analysis reported above showed that tracing at class level can save an order of magnitude of input effort while introducing only 15-30% more false positives compared to tracing at method level. So, regarding our case studies, one could argue that the tracing quality at the class level is relatively close to tracing at method level. Yet, we showed above that most of the quality problems when tracing at class level came from low-strength traces. This opens up an opportunity to investigate the option of automated filtering support (strength thresholds) for the trace candidate validation step to eliminate error-prone low-strength traces – thus improving the overall quality of the trace analysis at very little extra cost.

We experimented with three different kinds of filters, namely:

- *Threshold*: This filter eliminates all traces with strength lower than x. We applied this filter with different strength values.
- *Constant strength reduction*: This filter reduces the strength of each trace candidate by a constant.
- *Linear strength reduction*: The strength values of traces with 100% percent strength are not reduced while traces with strength of 0% are reduced with a maximum value (e.g., 10). The strength of traces with a strength value between 0% and 100% are reduced by a linear fraction of the maximum value.

Figure 37 depicts that all filters have the most effect on weaker traces (very left side of graph), whereas strong traces (right side) are hardly affected. We found the last filter, a scaled threshold, to be most effective although the others were good also. This filter did not use a constant threshold but one relative to the traces' strengths – the weaker the trace link, the stronger the filter. The aim of this filter was to leave strong traces strong but eliminate weak ones.

While filters are cheap, they do have a side effect. The trace analyzer does not generate false negatives, i.e., if the trace analyzer does not find a trace between two requirements, then there is none. Yet, the filters eliminate traces rather randomly with the assumption that most weak traces are false traces. Thus, a side effect of using filters is that they also eliminate some true traces. In other words, using a filter reduces the number of false positives but it introduces false negatives.

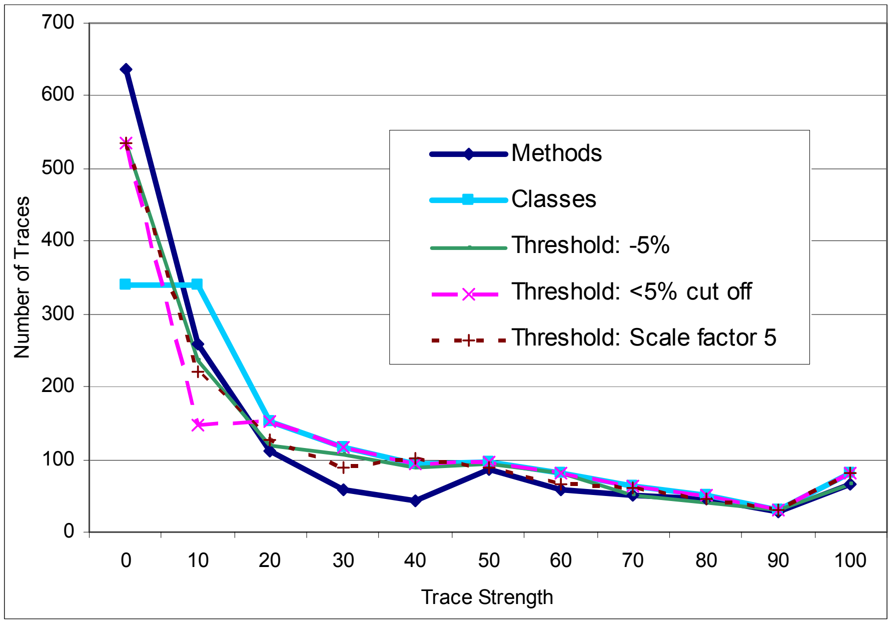


Figure 37. Eliminating false positives at class level w/thresholds [36]

Yet, this effect has advantages because many more false positives are eliminated than false negatives are added (with small filters). That is, the weakest 10% of the trace links contained only 1% of the true traces (i.e., observe that 26% false positives may be reduced to 14% false positives by only adding 1% of false negatives). Also, some applications that consume trace links may be more amenable to false negatives than false positives. For example, consistency checking is more confused by false positives than false negatives. Thus, filtering may be used to alter the quality effect in favour of the quality needs of the consumers of trace links.

5.3.4 Conclusion and Further Work

In this paper we have shown that cost-quality trade-offs play an important role when introducing traceability techniques and tools. Using three case studies we have demonstrated that it can be worthwhile to reduce the level of detail during tracing to save effort while the loss in quality might still be acceptable. Our empirical study results indicate that trace analysis on method level is roughly 10 times more expensive but produces almost no additional strong traces.

Further, we found that strong traces are very likely true traces whereas weak traces are very likely false traces. That is, the weakest 10% of all traces contain over 90% of false positives and only few true trace links. Thresholds thus can quickly eliminate false positives at the expense of also eliminating a few true trace links.

The recently defined paradigm of value-based software engineering [12][10] brings a new view into the trace analysis research area. Taking a value-based perspective can help save cost and by emphasizing investing effort on software artifacts with a perceived higher stakeholder value. Some initial results have been reported that consider value aspects in requirements traceability. However, these approaches have not conducted a cost-quality analysis to find out when and how intensive tracing in a specific context is worthwhile.

5.4 The Impact of Trace Correctness Assumptions on the Cost-Benefit of Requirements Tracing

Requirements tracing (RT) refers to the ability to follow the life of a requirement in forward and backward directions [51], e.g., by explicitly capturing relationships between requirements and related artifacts. For example, a trace between a requirement and a piece of source code can indicate that the code piece implements the requirement. Such traces support important applications, e.g., change impact analysis [133], which consists of identifying artifacts that have to be adapted when single requirements change.

Establishing traceability in a project increases project effort and needs to be justified based on solid data. However, the costbenefit of RT depends on multiple parameters, e.g., the granularity and correctness of traces, as well as on the tracing approach in a project, e.g., no tracing at all, ad-hoc-tracing, or some level of systematic tracing. Preliminary empirical studies mainly focused on the effects of trace granularity on RT cost-benefit [5][12][40]. For example, tracing requirements into source code can be done at method, class, or component level. The alternatives are associated with different levels of effort to establish traces, and also different levels of benefits, e.g., as method-level tracing provides a more precise basis for change impact analysis than class-level or component-level tracing.

The effect of trace correctness on RT cost-benefit is an important parameter (keeping traces correct over time can take considerable effort, while incorrect traces can significantly diminish the potential benefits of tracing) and its impact needs empirical evaluation. Such an evaluation is necessary to optimize tracing approaches for future projects.

In this work we provide an overview of a cost-benefit model for requirements tracing for the application "Change impact analysis" and address the influence of trace correctness on RT cost-benefit by providing an evaluation concept. We further analyze the consequence of selecting a trace alternative (no tracing, ad-hoc tracing, or systematic tracing) on RT cost-benefit.

The next section provides a cost-benefit model that explains all RT-relevant parameters, the influence of trace correctness, and the tracing alternatives. After that we propose an evaluation concept. The paper closes with a conclusion and an overview on further work.

5.4.1 Tracing Cost-Benefit Model

The RT cost-benefit model is an initial explorative model that builds on the cost and benefit parameters listed in Figure 38 and explained below. A focus is on the trace correctness parameter which is described in detail below.

Trace generation Cost Parameters

The costs of requirements tracing mainly depend on the effort for trace generation and trace rework. Trace generation effort grows with the number of requirements and artifacts to be traced, with more detailed granularity of traces (e.g., tracing into source code at component, class, or method level), later establishing of traceability (due to increased effort and risk to retrieve historic implementation details). On the other hand side, trace automation can reduce the effort for trace generation.

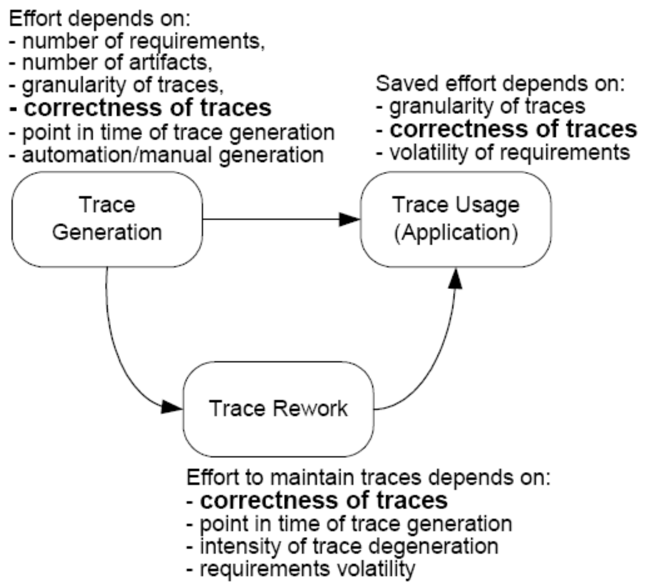

Effort depends on:
- number of requirements,
- number of artifacts,
- granularity of traces,
- correctness of traces
- point in time of trace generation
- automation/manual generation
Saved effort depends on:
- granularity of traces
- correctness of traces
- volatility of requirements
Trace Generation
Trace Usage (Application)
Trace Rework
Effort to maintain traces depends on:
- correctness of traces
- point in time of trace generation
- intensity of trace degeneration
- requirements volatility

Figure 38 Cost-benefit parameters for requirements tracing

The effort for trace rework is caused by the fact that captured traces typically degrade over time with changes of the traced artifacts, as the software product evolves (trace degradation).

Thus, trace rework is characterized by the effort to keep trace sets correct and consistent with evolving artifacts over time. It mainly depends on the frequency of trace generation and on the volatility of requirements and artifacts.

Trace Benefits to Change Impact Analysis

The benefit of traces can be characterized as the saved effort and less delay for change impact analysis when using traces. The benefit of traces depends on granularity of traces, as defined in [36]; higher trace granularity allows more precise change impact analysis and saves effort during the latter, because if change locations have to be identified at method level, traces at method level point directly to the change locations, whereas traces at class level need to be analyzed further. Furthermore, the benefit of a trace is the higher, the more frequently a requirement changes, as the associated traces are needed more often for change impact analysis.

Research Question – The Impact of Trace Correctness on RT's Cost-Benefit

Trace correctness describes the share of wrong or missing traces in a set of traces (false positives and false negatives), e.g., 100% correctness means that all existing dependencies between requirements and artifacts have been identified and that no wrong traces have been captured. On the other hand, a trace correctness lower than 100%, e.g., 60% means that 40% of the captured traces are either captured wrongly or correct traces have not at all been captured As you can see in Figure 1, trace correctness influences both the cost of trace generation and rework effort as well as the benefit of traces for change impact analysis.

Thus, concerning usage of requirements tracing for change impact analysis, we pose the following research question:

RQ: How does the correctness of requirements traces influence the value of traces for change impact analysis?

The value of traces for change impact analysis depends whether the attainment of a certain trace correctness causes more trace generation and rework effort than saved efforts for change impact analysis (in comparison to change impact analysis without traces) or the other way around. E.g., if the effort to generate and maintain traces (at a certain level of correctness) is higher than the effort for change impact analysis without traces, the value of these traces is 0 or lower. We assume that 100% correct traces provide the maximal trace value, because they maximize the saved effort for change impact analysis. But they are hard to realize in practice. The question is how this value of traces increases with a rising degree of trace correctness (see schematically in Figure 39).

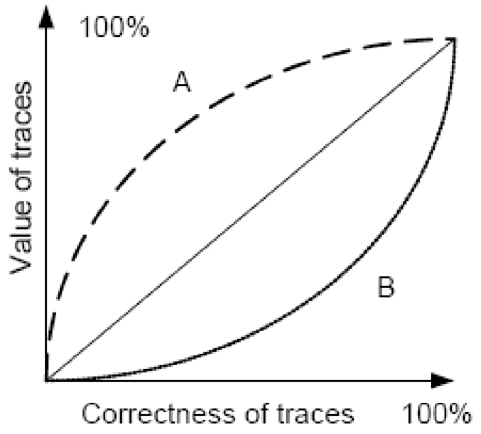


Figure 39 Trace Correctness vs. Trace Value

Figure 39 illustrates possible relationships of trace correctness to the “value” of traces. The goal of our evaluation is to find out if this relationship for a given trace application, such as change impact analysis, is likely to be linear; whether a low degree of trace correctness already provides considerable value for change impact analysis (curve A in Figure 39); or whether the correctness of traces has to be rather high before they can provide significant value (curve B).

The impact of trace correctness on trace cost and benefit also depends on the chosen tracing alternative, because each alternative reacts to trace degradation differently and the alternatives differ with their costs over the software life cycle, and provide different benefits. There are, in principle, 2 strategies to perform RT in a project:

- Ad-hoc Tracing captures traces during change impact analysis, e.g., when a single requirement changes, the engineer identifies all artifacts that have to be adapted and captures these relations between requirement and artifacts explicitly as traces.
- Systematic Tracing traced all requirements systematically to related artifacts before a trace application. The effort for trace identification and maintenance can be significant, because every requirement is traced. On the other hand capturing traces before their application is cheaper, because the effort to understand relations between requirements and artefacts, e.g., source code, and to capture them is easier during artefact creation than later on.

From experience we assume the impact of trace correctness on cost-benefit to vary with chosen trace strategy (see Table 24). E.g., trace generation effort with 100% trace correctness causes only medium generation costs with ad-hoc tracing, because only the requirements affected by a change request will be traced, whereas the effort for systematic tracing is high, because all requirements are traced with 100% correctness. Furthermore the trace benefit is only medium for ad-hoc tracing, because, traces are not complete, whereas with systematic tracing, the benefit is maximal.

Table 24 Influence of trace correctness on RT cost-benefit

	Ad-hoc Tracing	**Systematic Tracing**
100% correctness		
Trace generation cost	Medium	High
Trace rework cost	Medium	High
Trace benefit	Medium	High
65% correctness		
Trace generation cost	Low?	Medium?
Trace rework cost	<Medium?	High?
Trace benefit	<>Medium?	Medium?

Concerning tracing with reduced trace correctness, e.g., 65% (we assume a correctness under 65% to void all benefits, because the effort for trace rework will be much higher than the saved effort when using the traces for change impact analysis), we assume the trace generation effort with ad-hoc tracing to be low, because not all traces for the changing requirements have to be identified.

The benefits of tracing with 65% trace correctness are marked with a "?" because these are the measures we want to evaluate by an empirical study. The outcome of the evaluation (which is described below) can provide: for each tracing strategy efforts for trace generation and trace rework (costs) and the saved efforts (benefits) during change impact analysis, for both maintenance of a 100% correct trace set and a 65% correct trace set. From several empirical studies the goal is to provide an experience base that can be used for a new project to predict the optimal tracing strategy with an optimal level of correctness.

5.4.2 Evaluation Concept

We propose to evaluate the correctness of traces and the impact of trace correctness on RT cost-benefit with data from a case study where team members use the different tracing strategies (no tracing, ad-hoc tracing, and systematic tracing).

Further we will characterize the cost-benefit parameters described above (requirements and artifacts (pieces of source code, test cases, design components) to be traced). We will record all change requests during the project and gather the effort data for the three tracing strategies to compare overall effort variations coming from (a) the three strategies and (b) from different levels of trace correctness.

Further we will elicit feedback from the project participants on their perception of the usefulness of traces for change impact analysis and the related effort for creating and maintaining these traces. Based on the result of this initial case study we can calibrate the cost-benefits model and suggest further empirical studies with systematically variying context parameters.

5.5 Integrated Developer Tool Support for More Efficient Requirements Tracing

Global software development has contributed an increasing share to big software development companies like Siemens IT Solutions and Services (SIS PSE) business in the last years [130]. In global software development projects the overview of the project manager and key developers on requirements and the artifacts that implement them become more crucial as a) these requirements and artifacts tend to evolve concurrently and b) in distributed projects there is often less informal personal contact that helps to synchronize the views of project participants. To make up for the potential loss of information project managers need to make communication channels between developers and to management more effective.

Software project managers perform multiple tasks that are greatly facilitated by complete and correct requirements-related information such as: a) compliance/consistency checks at the transition from one phase to the next, e.g., from analysis to design, where the project manager has to check that each requirement is actually considered by certain design elements; b) acceptance testing where the project manager has to check that each requirement has been properly implemented and tested; and c) change impact analysis where the project manager analyzes the likely impact of a requirement change based on the identification of potential change locations in artifacts like design, source code,or test cases.

These tasks have in common that a project manager needs sufficient information about dependencies between requirements and other artifacts (typically design, source code, or test cases) to perform them properly. Requirements tracing (RT) is a technique to explicitly capture and maintain these dependencies (so-called traces). RT is demanded by software engineering standards like CMMI [102]. For example, a trace between a requirement and a source code element, e.g., a source code method, indicates that the method implements this requirement.

A project manager can use these traces, e.g., during change impact analysis, to effectively and efficiently identify potential change locations in source code for each changed requirement.

The availability of complete and correct traces can save considerable effort for change impact analysis, compared to the case without traces, where the project manager (or responsible developer) would have to check all potential source code elements for possible change impact matches. However, despite the benefits to trace applications like change impact analysis traditional tracing approaches have several drawbacks:

- The effort to manually capture and maintain traces can be considerably high, which in practice often lowers the completeness and sometimes the correctness of available traces;
- In practice capturing traces during coding typically does not work. Instead, a typical tracing process would be: coding first, capturing traces later, e.g., when some packages have been finished. The longer the time between coding and tracing the more incomplete or even wrong traces tend to be as developers might not remember details anymore and the high effort of tracing reduces their motivation to capture tracing after coding.
- Approaches for the automated generation of traces often do not provide sufficient trace quality; as traces may identify false dependencies between artifacts, or vital traces may be missing; thus, practitioners learned not to fully trust the results of these approaches alone.
- Developers are oftenreluctant to use a new documentation tool that is not part of their typical tool set (“yet-another-tool syndrome”), e.g., a requirements management tool to capture and maintain traces.

While there are many methods and techniques on how to technically store requirements traces, I found very few approaches that sufficiently consider the issues mentioned above. I propose a RT approach that is integrated into the developer tool set and addresses the issues mentioned above as follows: (a) The plug-in-based tool automatically imports a list of requirements from a requirements management tool, such as Requisite Pro, into an integrated development environment (IDE), e.g., Eclipse, where developers efficiently can create traces between requirements and source code elements (methods or classes) directly in their familiar work environment. Thus, there is no need for developers to use an extra tool in addition to their IDE or to look up references in external requirements documents; (b) developers can then create traces simply by marking a requirement from the imported list. The requirement is then automatically inserted as a reference into the comment header of the currently selected method or class; (c) finally, the captured traceability information is automatically re-imported into the requirements management tool where project managers can view the traceability information in a traceability matrix or traceability tree and can use this information for change impact analysis.

From the integrated tool support for RT I expect the following benefits: a) less cognitive complexity for developers to capture and maintain traces as they continue to work within their familiar work environment; b) less delay of trace generation as developers can create and maintain traceability information easily without disruption during development (process-driven tracing); c) more complete trace generation: by reducing the effort of trace generation the acceptance of developers to capture and maintain traces is expected to increase; d) more correct trace generation: due to less delay, the captured trace information should be more correct than with an approach where code is developed first and traces are captured later on; further, enabling faster quality assurance of traces and making trace capture progress more visible can have a positive impact on developer motivation; and finally e) less effort of RT as the plug-in approach automates the development of cross-references and traceability matrices and thus reduces the effort considerably. These effects can reinforce each other due to a positive feedback cycle between trace generation and trace usage like change impact analysis (CIA) (see Figure 40).

Overall, the tool supported RT approach is expected to provide more correct and more complete traces with lower effort in comparison to other manual RT approaches and thereby is expected to considerably improve the efficiency of requirements change impact analysis capabilities in distributed software development projects. If traces were fully correct and complete, the impact analysis effort for changed requirements would be reduced to locating potentially affected code elements by following the traces from these changed requirements, instead of searching the whole source code.

Change impact analysis (CIA) is a task that can benefit from effective RT. During CIA a project manager analyzes for a change request which artifacts could be possibly affected by modifications to implement this change request. Concerning source code, this means that somebody has to identify the change locations in the source code to implement the change request. Traces leverage CIA, because these change locations would be easy to find by following correct and complete traces. Nevertheless, there are multiple factors that influence RT.

5.5.1 An Initial RT Effects Model

Figure 40 depicts a model of factors of RT that influence the value of traces for trace applications like change impact analysis (CIA).

There are three major blocks of factors: (1) project setting, which contains project characteristics (number of requirements, number of artifacts to be traced) that influence tracing efforts, eventually tools to be used, etc.; (2) trace generation, which consists of the main factors that influence (3) trace quality. Finally, trace quality determines, how good project managers can use the captured traces for trace applications like (4) CIA. The factors and its dependencies are described in detail in the following subsections.

Project Setting. The number of requirements is a basic factor that defines the general order of magnitude of tracing. It might also influence the number of artifacts that emerge during the project and thereby the number of potential dependencies between requirements and artifacts that need to be captured as traces, e.g., the more requirements exist the higher is the number of source code elements, e.g., source code classes, and the higher is the number of traces to be captured in order to assure a complete trace set.

The characteristics of a project like number of requirements and number of artifacts determine the tracing process and tool selection decision, i.e. the way of how traces should be captured in this project. For example, if the number of requirements is low (in a small-scale project with a handful of requirements), using a sophisticated requirements management tool might be an overkill.

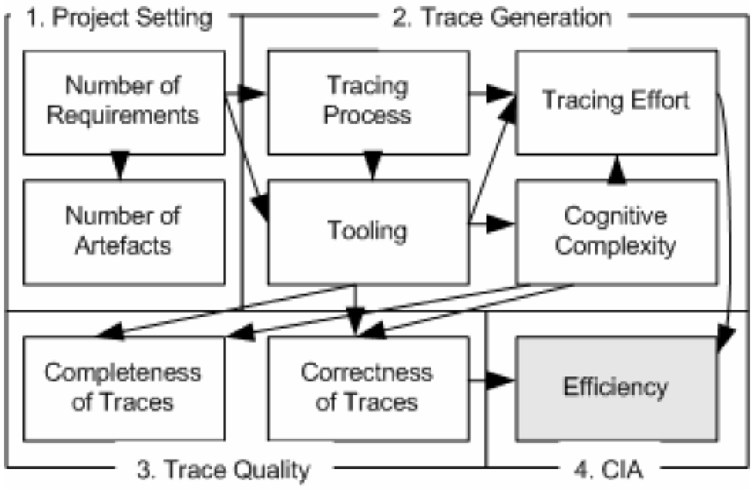


Figure 40 Model of factors that influence requirements tracing for change impact analysis

Furthermore, establishing explicit traceability might also be too expensive in small-scale projects, because the overview of dependencies could be kept informal (no explicit traces) due to the simplicity (small size) of the project. Thus, a tracing process for change impact analysis would rather be ad-hoc than systematic. (see tracing process).

Trace Generation. The tracing process defines when traces should be captured. For example, by using an integrated tool-based approach, developers can capture requirements immediately during coding (continuous tracing), thereby reducing the delay of capturing traces. With other tools, coding has to be finished first and traces can be captured afterwards. Another tracing process alternative would be ad-hoc tracing: when no traces exist, the project manager has to identify dependencies (traces) ad hoc during CIA.

A good tracing process reduces delay and cognitive complexity, as it is often easier for a developer to capture the right traces correctly when the time between artifact creation and capturing traces is short. Delay is the time between artifact creation and capturing of traces. The higher the delay, the worse the likely impact on correctness and completeness of traces, because it becomes increasingly harder for developers to remember certain dependencies between artifacts the longer the time since the creation of these artifacts. Less delay also facilitates the quality assurance of traces (and tracing progress), which in turn can have a positive feedback effect on developer motivation to completely and correctly capture their traces.

Developers have to use certain tools to capture and maintain traces between requirements and source code. They might use requirements management tools, Excel, or the proposed Eclipse plug-in. As mentioned above, each tool provides its own mechanisms to capture traces. Good tool support reduces both cognitive complexity and delay of tracing, e.g., by allowing developers to easily capture traces during coding instead of capturing them later or in a separate requirements management tool. Each tool can be assessed from a usability perspective (cognitive complexity), e.g., the way how traces are captured. For example, with an Excel sheet, the developer has to insert a mark in a cell that connects a requirement and a source code element to create a trace, whereas in requirements management tools other mechanisms for capturing traces are provided. With our plug-inbased approach, the capture of traces is fully integrated into the developer's work environment without the need to use other tools.

Furthermore, traces are created just be marking requirements in a source code method or class. Thereby the cognitive complexity and effort of tracing are reduced. Cognitive complexity can be related to the number of tools needed concurrently for tracing (development environment and requirements management tool vs. just the development environment). The lower cognitive complexity, the lower is the effort for trace generation (e.g., minutes to capture a trace), because switching between tools and filling-in attributes is likely to cause more effort than marking requirements directly in the development environment.

Trace Quality. The main factors of trace quality are completeness and correctness of traces. Completeness means that all existing traces are captured on a certain level of precision (such as source code classes or methods), e.g., all classes in the source code have traces back to requirements (we have discussed precision intensively in [36][41][63]) . Correct traces means that only valid traces are captured: if a requirement traces to a class that does not implement (part of) the requirement, the trace is considered incorrect.

The more correct and complete traces are, the higher is their value for change impact analysis, because, correct and complete traces allow the project manager to trustingly follow the traces to find all possible change locations regarding a given change request.

Efficiency of Change Impact Analysis (trace usage). The integrated way of tracing and the reduced effort are expected to lead to in-time, more correct and complete traces, which is likely to increase the value of traces for, e.g., project managers, when they want to use traces for particular tasks like change impact analysis.

Generally, the lower the cognitive complexity and delay of capturing traces (which depends on the available tool support and the chosen tracing process), the lower is the average effort to capture traces, which increases efficiency of tracing (average effort to capture a trace), and the higher the developers' acceptance of tracing. The lower the effort (and the higher developers' acceptance), the more willing they can be expected to be to capture traces; which increases trace effectiveness (number of captured traces). The increases in effectiveness and efficiency reduce the risk of having incorrect and incomplete traces, because the probability to miss traces is reduced, and the loss of saved efforts during CIA is reduced.

5.5.2 Tracing Approaches for Change Impact Analysis

The Support Center Configuration Management (SCCM) at SIS PSE provides projects with configuration and requirements management tooling and process consulting and support and therefore has experience about tracing and Change Impact Analysis (CIA) approaches used in practice. As mentioned above, tracing is still done mostly manually, as automated trace generation tools (e.g., academic approaches) have not made their way through to practice yet. The approaches range from: (a) CIA without explicit tracing, (b) CIA with tracing by a requirements management tool, to (c) CIA with continuous tracing by the RT plug-in.

Ad-hoc: CIA without explicit tracing

With this approach no explicitly captured traces are used to support CIA. Instead, the project manager has to check for each changing requirement which source code elements are affected. Thus, he does not use a tracing tool, and the effort to identify traces ad-hoc is expected to be high due to the delay between coding and tracing.

Systematic: CIA with tracing by a REQM tool

The requirements are stored in a requirements database, e.g., in Requisite Pro. In order to create traces between requirements and source code, source code element references have to be created in Requisite Pro, e.g., for each source code method there should be an element in Requisite Pro. When all source code elements are represented in Requisite Pro, the requirements engineer can manually fill in traceability information by filling-in the “tracesto/from” attribute of each requirement. The systematic tracing process means that traces are captured in intervals, e.g., when a component has been finished; so there usually is a delay between coding and tracing.

Continuous: CIA with tracing by RT plug-in

Using a plug-in that facilitates easy and fast tracing within the developer IDE (see Section 4 for details) reduces the cognitive complexity, because developers do not have to use other tools for tracing than their usual work environment. Furthermore, this approach supports continuous (or process-driven) tracing, where developers can trace “by the way” during coding, so that the delay between coding and tracing can be significantly reduced. This should positively affect the correctness and completeness of traces.

5.5.3 Research Issues

As explained above, requirements tracing can be a good support for CIA. However, the effort to capture traces is the biggest obstacle to get complete and correct traces, which are the prerequisites for efficient CIA. Three particularly relevant tracing approaches (from a practitioners’ point of view) differ in the tools they use and the processes how and when traces are captured: the Ad-hoc approach uses no tools and does not capture traces in advance to CIA, the Systematic approach uses requirements management tools to capture traces systematically in intervals, and the Continuous approach uses the plug-in and the IDE to capture traces. Given the project setting factors (from figure 1), which are the same for all approaches, like number of requirements, number of artifacts, and the number of traces to be captured, the research questions can be stated as follows:

Research Issue 1: How much does the plug-in-based continuous tracing approach reduce effort (in working hours) of trace generation in comparison to the other alternatives (ad-hoc and systematic)?

One advantage of the integrated tool support in comparison to the REQM alternative is the creation of source code elements references: the plug-in fully automates this step, whereas methods and classes have to be manually inserted into Requisite Pro as prerequisite to capture traces later on.

Research Issue 2: How much does the reduced effort, the reduced delay of our plug-in based continuous tracing approach improve the completeness (in % of existing dependencies) and correctness (false positives in % of captured traces) of captured traces compared to the alternatives?

Due to the reduced effort and simple way of tracing, the developers are enabled to capture more traces (completeness). Concerning correctness, the only source of error for a developer is to mark a wrong requirement. The alternatives provide more opportunity for error, e.g., typing or navigation errors. Thus, the lower effort of tracing and increased user acceptance should result in a set of more correct and more complete traces.

5.5.4 A Plug-In Tool for Continuous Requirements Tracing

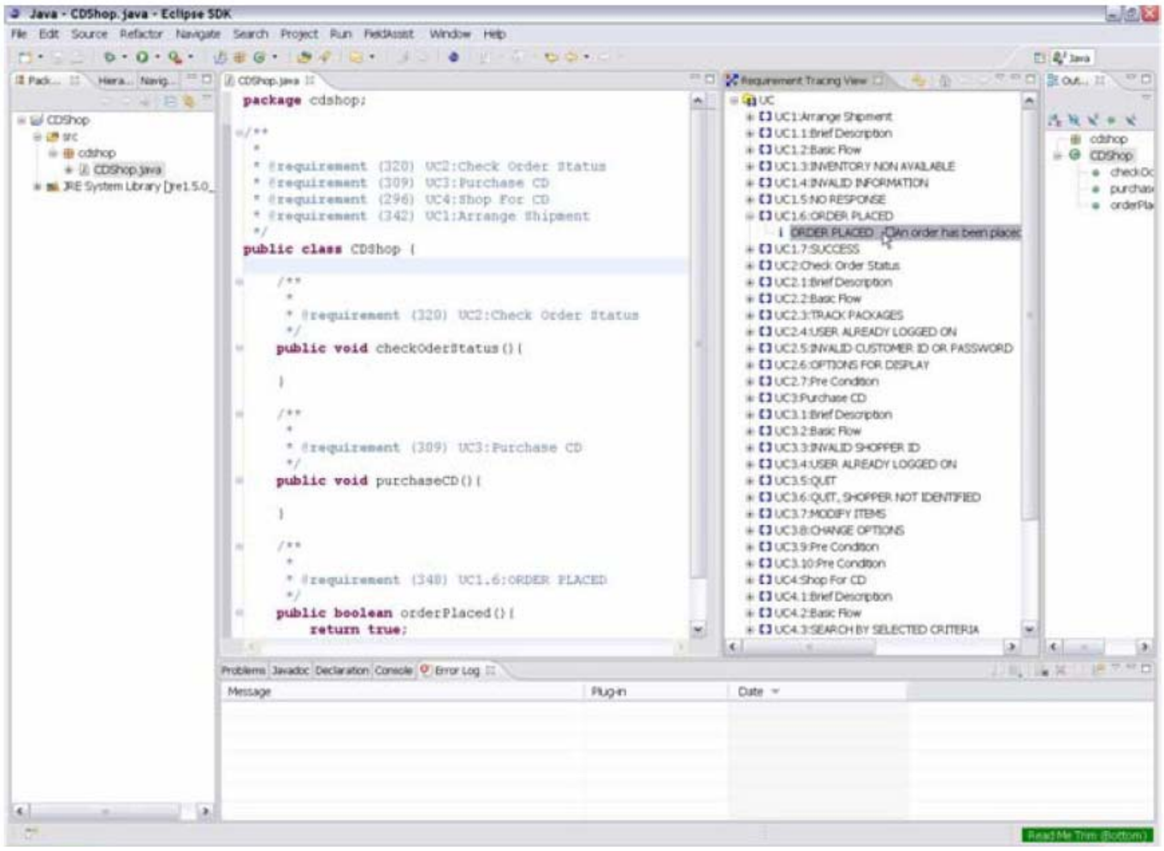
Plug-in-based requirements tracing is an approach that aims at improving tracing from requirements to source code elements by automating sub-tasks of tracing and thereby significantly reducing tracing efforts.

The developers get the requirements they are to implement and use an integrated development environment (IDE), e.g., Eclipse or Visual Studio. Concerning requirements tracing, they would usually create some Excel matrices to store traceability information or draw a license for the project's requirements management tool, where they could insert the traceability information by creating traceability views. After the developers have created their contributions to traceability information, the information from all team members would come together at the project manager who would collect the full traceability information from all development teams. He could then use this information to analyze the likely change impact of a given change request.

Using the integrated tool support for tracing (the RT Plug-in) a developer who writes code in Eclipse should be able to link the requirements that are captured and maintained in a requirements management tool (Requisite Pro) to the code elements he produces. That means:

- Requirements identifiers, titles and short descriptions are exported from Requisite Pro and displayed in the Eclipse plug-in as a list.
- A developer who works on a source code class or method just sets the cursor into a method or class and marks the related requirement(s) in the list to create the traceability link.
- The requirements identifier is inserted in the javadoccomment of the class or method; when the source code file is stored, the traceability information is imported into Requisite Pro, allowing the trace links between requirements and source code elements to be displayed in the Requisite Pro traceability matrix.

Users of the RT plug-in are software developers using Eclipse to work on source code. Project managers using Requisite Pro for requirements management (RM) can retrieve the traceability information (which requirement is implemented where in source code) from within Requisite Pro. Thus each role has the advantage of having the relevant trace information in its native tool (IDE or RM tool).



```
package cdshop;

/**
 *
 * @requirement (320) UC2:Check Order Status
 * @requirement (309) UC3:Purchase CD
 * @requirement (296) UC4:Shop For CD
 * @requirement (342) UC1:Arrange Shipment
 */
public class CDShop {

    /**
     *
     * @requirement (320) UC2:Check Order Status
     */
    public void checkOderStatus(){

    }

    /**
     *
     * @requirement (309) UC3:Purchase CD
     */
    public void purchaseCD(){

    }

    /**
     *
     * @requirement (348) UC1.6:ORDER PLACED
     */
    public boolean orderPlaced(){
        return true;
```

```
UC
  UC1:Arrange Shipment
  UC1.1:Brief Description
  UC1.2:Basic Flow
  UC1.3:INVENTORY NON AVAILABLE
  UC1.4:INVALID INFORMATION
  UC1.5:NO RESPONSE
  UC1.6:ORDER PLACED
    ORDER PLACED -An order has been placec
  UC1.7:SUCCESS
  UC2:Check Order Status
  UC2.1:Brief Description
  UC2.2:Basic Flow
  UC2.3:TRACK PACKAGES
  UC2.4:USER ALREADY LOGGED ON
  UC2.5:INVALID CUSTOMER ID OR PASSWORD
  UC2.6:OPTIONS FOR DISPLAY
  UC2.7:Pre Condition
  UC3:Purchase CD
  UC3.1:Brief Description
  UC3.2:Basic Flow
  UC3.3:INVALID SHOPPER ID
  UC3.4:USER ALREADY LOGGED ON
  UC3.5:QUIT
  UC3.6:QUIT, SHOPPER NOT IDENTIFIED
  UC3.7:MODIFY ITEMS
  UC3.8:CHANGE OPTIONS
  UC3.9:Pre Condition
  UC3.10:Pre Condition
  UC4:Shop For CD
  UC4.1:Brief Description
  UC4.2:Basic Flow
  UC4.3:SEARCH BY SELECTED CRITERIA
```

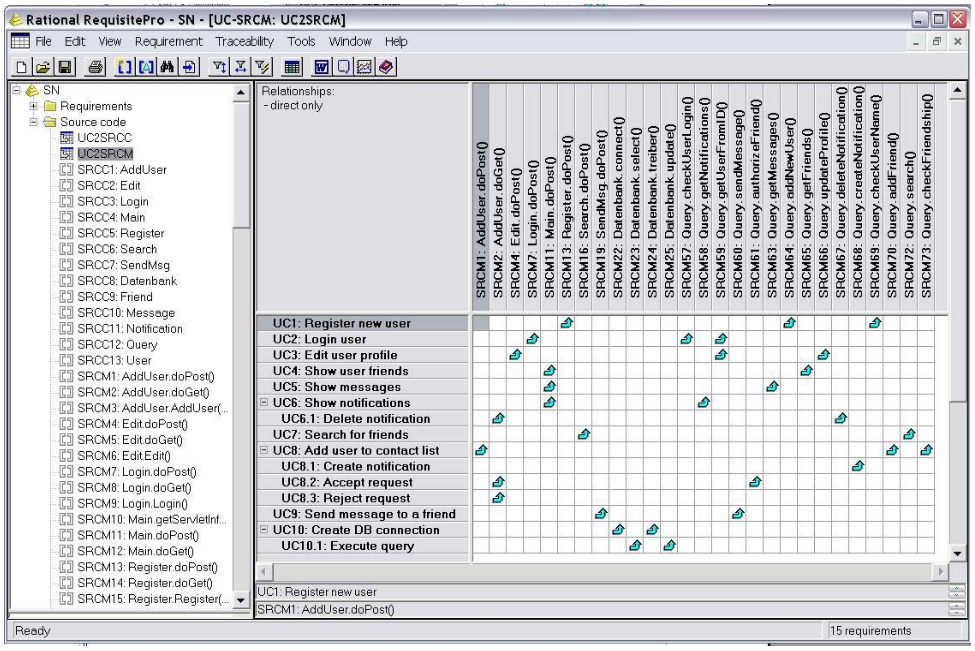
Java - CDShop.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run FieldAssist Window Help
CDShop
src
cdshop
CDShop.java
JRE System Library [jre1.5.0_
CDShop.java
Requirement Tracing View
Problems Javadoc Declaration Console Error Log
Message
Plug-in
Date
cdshop
CDShop
Read Me Trim (Bottom)

Figure 41 The Requirements Tracing Plug-in for Requisite Pro

The requirements from Requisite Pro can be imported into Eclipse without opening Requisite Pro. Furthermore, a developer can select a checkbox “notify when a method/class has no requirements trace”. When the developer selects this checkbox, he gets a warning message during closing the project, if not all methods (or class) trace to at least one requirement.

The requirements are shown in a list and they are grouped according to their types. For each requirement in the list, the unique number (identifier in Requisite Pro) and name is displayed. The short description of the requirements also is displayed via an expandable tree structure (so that the description is not displayed by default, but can be expanded on demand).

Furthermore, there is a refresh button in the requirement view. Whenever a new requirement is added in Requisite Pro, the list in Eclipse should be refreshed by a pressing this button. When the requirements are imported and the requirement view is opened, the developer has the possibility to link certain requirements from the list to particular pieces of source code. A trace-link can be deleted by deleting the line with the requirement information in the javadoc-comment. *Figure 41* shows the following examples: The requirement “Check order status” with the unique ID “320” is traced to the method “checkOrderStatus”. The class “CDShop” where the method is located carries all requirements that are traced to methods of the class in its javadoc header (4 methods). When the code file is saved or the editor window is closed, the traceability information is automatically exported and re-imported into the defined Requisite Pro project. An extra package, named “Source Code” will be created in Requisite Pro, where all the code elements and traceability views will be automatically created. Each source code entity (method or class) that was created in Eclipse is imported into this package in Requisite Pro. Source Code classes are captured in Requisite Pro as requirement type named Source Code Class (SRCC). Source Code methods are captured in Requisite Pro as requirement type named Source Code Method (SRCM). These types are created automatically when the traceability information is re-imported into Requisite Pro.



SN
Requirements
Source code
UC2SRCC
UC2SRCM
SRCC1: AddUser
SRCC2: Edit
SRCC3: Login
SRCC4: Main
SRCC5: Register
SRCC6: Search
SRCC7: SendMsg
SRCC8: Datenbank
SRCC9: Friend
SRCC10: Message
SRCC11: Notification
SRCC12: Query
SRCC13: User
SRCM1: AddUser.doPost()
SRCM2: AddUser.doGet()
SRCM3: AddUser.AddUser(...
SRCM4: Edit.doPost()
SRCM5: Edit.doGet()
SRCM6: Edit.Edit()
SRCM7: Login.doPost()
SRCM8: Login.doGet()
SRCM9: Login.Login()
SRCM10: Main.getServletInf...
SRCM11: Main.doPost()
SRCM12: Main.doGet()
SRCM13: Register.doPost()
SRCM14: Register.doGet()
SRCM15: Register.Register(...

	SRCM1: AddUser.doPost()	SRCM2: AddUser.doGet()	SRCM4: Edit.doPost()	SRCM7: Login.doPost()	SRCM11: Main.doPost()	SRCM13: Register.doPost()	SRCM16: Search.doPost()	SRCM19: SendMsg.doPost()	SRCM22: Datenbank.connect()	SRCM23: Datenbank.select()	SRCM24: Datenbank.treiber()	SRCM25: Datenbank.update()	SRCM57: Query.checkUserLogin()	SRCM58: Query.getNotifications()	SRCM59: Query.getUserFromID()	SRCM60: Query.sendMessage()	SRCM61: Query.authorizeFriend()	SRCM63: Query.getMessages()	SRCM64: Query.addNewUser()	SRCM65: Query.getFriends()	SRCM66: Query.updateProfile()	SRCM67: Query.deleteNotification()	SRCM68: Query.createNotification()	SRCM69: Query.checkUserName()	SRCM70: Query.addFriend()	SRCM72: Query.search()	SRCM73: Query.checkFriendship()
UC1: Register new user						↗													↗					↗			
UC2: Login user				↗									↗		↗												
UC3: Edit user profile			↗												↗						↗						
UC4: Show user friends					↗															↗							
UC5: Show messages					↗													↗									
⊟ UC6: Show notifications					↗									↗													
UC6.1: Delete notification		↗																				↗					
UC7: Search for friends							↗																			↗	
⊟ UC8: Add user to contact list	↗																								↗		↗
UC8.1: Create notification																							↗				
UC8.2: Accept request		↗															↗										
UC8.3: Reject request		↗																									
UC9: Send message to a friend								↗								↗											
⊟ UC10: Create DB connection									↗		↗																
UC10.1: Execute query										↗		↗															

UC1: Register new user
SRCM1: AddUser.doPost()

Figure 42 Traceability Matrix generated in Requisite Pro

For each requirement type in Requisite Pro, two traceability views are created automatically: one that shows the traces from the requirements to source code classes, and one to source code methods. Figure 42 shows the "Source Code" package with the created elements at the left pane; for example, the requirement "Check Order Status" traces to three methods in class "CDShop", namely "checkOrderStatus()", "purchaseCD()" and "orderPlaced()". Whenever traceability information is changed in Eclipse the updated traceability information is re-imported into Requisite Pro. The old traceability information of the concerning code elements is deleted automatically. Manual but tool supported tracing by REQM tools as it is current status quo in practice is expensive. The high effort of manual tracing influences negatively trace applications like change impact analysis. Traces support CIA because they save effort for analyzing related artifacts. But when the creation of traces is too expensive (like with REQM tools) the pay-off is marginal. With the plug-in the efforts for tracing should be reduced and thereby developers' motivation to trace should be increased, so that the completeness and correctness of traces improves. Finally, trace applications like change impact analysis get more efficient due to the usage of more complete and correct traces.

5.5.5 Comparison of Alternatives

In order to compare the plug-in-based continuous tracing approach with the alternatives, I observed a set of projects at Siemens PSE to evaluate the tracing efforts, correctness and completeness of each alternative. The projects were different in domain (transportation systems, telecommunication, etc.), similar in size: medium size projects, with 2 to 4 sites (e.g., Austria, Romania, Slovakia), and between 10 and 60 team members.

Concerning the project setting factors from our model in Figure 40, these projects can be characterized as follows:

- Number of requirements: between 150 and 300;
- Number of source code methods to be traced: 6,000 to 13,000;
- Number of traces per requirements: 80 to 270.

The numbers of requirements and source code methods were easy to count. For the number of traces per requirement, I counted the traces from representative random samples. Based on these project setting factors I compared the effort to trace requirements to source code methods, the completeness and correctness of traces for the 3 tracing alternatives described above. The 3 factors tracing effort, completeness, and correctness mainly determine the benefit of traces for change impact analysis. The ideal would be to have minimal tracing efforts resulting in a high completeness and correctness of traces.

Each tracing alternative differs has a specific tracing process and tools (which determine cognitive complexity) described in Figure 40 (trace generation factors).

CIA without explicit tracing (Ad-hoc)

This alternative serves as a baseline so that I can compare CIA efficiency of the CIA alternatives supported by tracing (the Systematic and Continuous alternatives) with a CIA variant without explicit tracing.

Tracing process and tooling

When a change request occurs, tracing is done ad-hoc that is the project manager checks for each requirement affected by the CR which source code elements are related to these requirements and also need to be adapted or at least analyzed. There are no traces captured in advance of a change request. No tools are used for tracing, requirements are usually stored in a text document; source code is developed in an IDE. The cognitive complexity is given by looking up relevant requirements in the word document and checking them with source code elements to find dependencies.

Tracing effort

Since no traces between requirements and source code methods are captured in advance of a change request, all source code elements have to be checked for potential impacts. Overall this results in a high tracing effort, especially when the delay between change request occurrence and coding is high. Since tracing is done ad-hoc and not during development, there is the risk of not remembering existing dependencies between requirements and source code elements, especially when a CR occurs very late after development. That means that the effort to identify related requirements and source code elements is very high, namely 2 minutes per trace on average. Given the fact that one requirement has between 80 and 270 traces into source code methods in our projects the total effort to trace only one requirement affected by a change request would be between 3 and 9 hours (see Table 25a).

Completeness and correctness of traces

The delay mentioned above also could have a negative impact on the correctness of traces. Concerning completeness, ad-hoc tracing will never reach systematic approaches or the like, because traces are only identified for given change requests and impacted requirements and never fully for all requirements. Thus, the completeness of traces that could be captured with ad-hoc tracing mainly depends on the change request rate of the project. In our class of projects the completeness never was larger than 30% (see Table 26).

CIA supported by Tracing with Requisite Pro (Systematic)

When requirements management tools are used, the common practice for tracing is to fill in traceability matrix views in these tools, e.g., Requisite Pro (which is besides Doors the most used requirements management tool within SIS PSE).

Tracing process and tooling

Developers trace requirements to source code methods in intervals, e.g., when a component has been finished). That means that there is a delay between coding and tracing, but shorter than with the ad-hoc tracing alternative.

Concerning tooling, the requirements are stored in Requisite Pro, which is used also for tracing; source code is developed in an IDE, e.g. Eclipse. Concerning the cognitive complexity of tracing, the developer produces code in the IDE, and for tracing has to use Requisite Pro (an extra tool, which often results in a “yet-another tool syndrome” [97]) for tracing.

Tracing effort

In order to trace requirements to source code methods, a developer has to create source code method references in Requisite Pro, has to create a traceability view, and finally has to trace requirements to source code methods in that view.

Overall, this results in a high tracing effort. From our project analyses the efforts to capture a single trace was approx. 15 seconds. This results in total tracing efforts of 50 to 167 hours for 150 requirements (depending on the # of traces per requirement), and 99 to 334 hours for 300 requirements (see Table 25b).

Completeness and correctness of traces

This high effort and the fact that developers have to use an extra tool for tracing often leads a low developer motivation and to the result that tracing is done incompletely (20 to 40% completeness; see Table 26). Furthermore, the correctness is also threatened by typos etc. when creating the matrices.

CIA supported by Plug-in-Based Requirements Tracing (Continuous)

The main goal of the plug-in based requirements tracing approach was to reduce tracing efforts by integrating tracing facilities into the developers’ working environment and thereby to improve trace completeness and correctness in comparison to the other alternatives by supporting continuous tracing during development.

Tracing process and tooling

As described in detail in the last section, developers trace requirements to source code during development via the plug-in.

Tracing effort

As requirements just have to be looked up in the list and be double-clicked in order to capture a trace, the effort typically is 2 to 3 seconds (on average 2.4 seconds in a feasibility study with practitioners at SIS PSE) per trace used to calculate the values in Table 25b.

Completeness and correctness of traces

I examined that both the lower complexity and the lower tracing efforts seem to lead to a higher developer motivation and thereby to higher completeness of traces (see Table 26). Correctness is also better than with the other alternatives, because the possibility to make errors is restricted to a wrong click on a requirement in the list.

Table 25a. Ad-hoc tracing effort.

	Tracing efforts (in working hours per req.)	
	Min	**Max**
Ad-hoc	3	9

Table 25b. Systematic and Continuous alternatives' effort.

	Tracing efforts (total in working hours)	
	for 150 requirements	**for 300 requirements**
Systematic	50 to 167	99 to 334
Continuous	8 to 26	16 to 53

Table 26. Comparison of tracing alternatives' qualities.

	Completeness	**Correctness (false positives in % of captured traces)**
Ad-hoc	5% to 30%	5-10%
Systematic	20% to 40%	10%
Continuous	60% to 75%	5%

The results of the comparison for the continuous plug-in based tracing approach are promising, because it seems to be a chance to improve requirements tracing in a systemic way by reducing the efforts and increasing completeness and correctness.

5.5.6 Discussion

Ad hoc tracing seems, at first sight, to be the cheapest alternative (compared with the systematic approach), as there are no costs for trace identification and maintenance. In projects where requirements changes are very likely this alternative might become very costly, because the project team or maintenance personnel have to do "tracing" ad hoc. The later these change requests happen, the more costly tracing gets during development. Further, tracing efforts on activities that are on the critical path for project or maintenance task completion will effectively delay the overall finish. Omitting tracing at that point incurs a high risk of lower-quality solutions and/or erosion of system design. The high effort of tracing seems to be the main reason why requirement tracing is seldom performed completely in projects.

Benefits of integrated tracing

Reduction of effort and delay – higher developer motivation

The continuous tracing approach by using the requirements tracing plug-in reduces tracing efforts and delay between coding and tracing significantly. A further strength of the continuous tracing approach is that it is integrated into the developers' working environment, so that they do not have to use other tools for tracing. Our observations show that 5 to 7 seconds per trace is a threshold for the developers to accept tracing tasks. That means that the reduced efforts (below the threshold), the integrated way of tracing and the short delay between coding and tracing have a positive effect on the developers' motivation for tracing, as depicted in Figure 43.

Our observations indicated developers' motivation to be a key factor for having complete and correct traces. If trace generation is too expensive (effort per trace higher than 5 to 7 seconds) and complex, the motivation to capture traces will be low, resulting in an incomplete and maybe incorrect set of traces. Finally, low quality of traces has a negative effect on trace applications like change impact analysis, because lower trace quality introduces added risk and/or more effort has to be spent to check potential change impacts in source code.

On the other hand, if tracing was sufficiently cheap and easy (2 to 3 seconds with an enabler such as the plug-in concept), the higher motivation to capture traces is likely to bring higher tracing completeness and correctness (see Table 26).

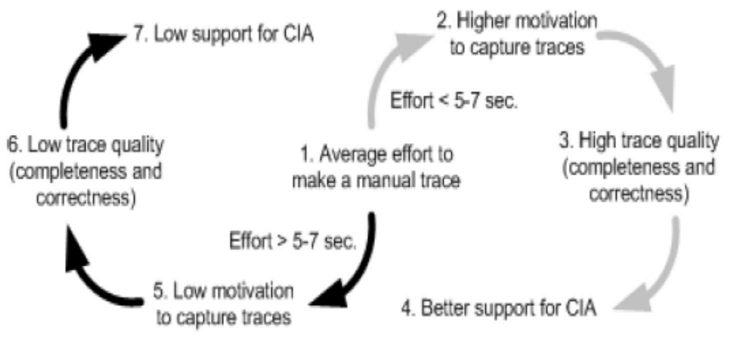

7. Low support for CIA
2. Higher motivation to capture traces
Effort < 5-7 sec.
6. Low trace quality (completeness and correctness)
1. Average effort to make a manual trace
3. High trace quality (completeness and correctness)
Effort > 5-7 sec.
5. Low motivation to capture traces
4. Better support for CIA

Figure 43 Positive and negative circles of requirements tracing

Furthermore, I discussed the plug-in-based approach with a panel of experts (requirements managers and developers) at SIS PSE and discussed the plug-in benefits and limitations. Both developers and requirements managers liked the approach very much due to its integration into existing work environments and the reduced effort of tracing. This indicates a high acceptance of the approach among a broad range of practitioner roles.

Integration into working environment of developers

One of the main intents of the plug-in-based RT approach was to consider the lessons learned from [97] when developing a requirements tracing solution. The lessons learned were:

- Provide smooth integration with existing tools: Developers typically have only little interest in using complex additional tools that are perceived as keeping them from doing their job. By fully integrating the RT approach into the development environment, the developers do not have to use additional tools.
- Make only small changes to work practices of developers. Tailoring the traceability strategy to the specific needs and development context is the key to success ("yet another tool" syndrome). By avoiding additional tools and giving the developer the means to very simply capture traces (they can do tracing "by the way"), the work practices of the developers change only slightly and the "yet another tool" syndrome is also avoided.

Higher completeness and correctness of traces

Concerning change impact analysis, the plug-in also promises improvements. Generally, without traces, developers would have to check the whole source code for change impacts, which results in a high effort. With fully correct and complete traces (the ideal case), the effort for checking the source code is significantly reduced, because developers just have to follow the traces to find the possible change impacts. With less than fully correct and complete traces, the effort is also reduced, but there is the risk that some possible change impacts are overlooked. The plug-in based approach reduces the effort and cognitive complexity of tracing, and thus should increase the developers' motivation for tracing, which in total increases the quality of traces (correctness and completeness) and makes the change impact analysis more efficient. This should be especially true for larger projects, where the amount of source code is very high and where traces could provide a significant increase of change impact analysis' efficiency.

Quality assurance of traceability

Our solution provides the means to automatically check if all methods in a class are related to requirements. If not, a warning can be displayed, containing the methods in source code that are not traced to a requirement, which is a good support to immediately check consistency and completeness both by the development team and the project manager.

Manual vs. automated trace generation

Practitioners seem to trust manual traces more (despite their known error-proneness) know the high effort to generate traces manually. On the other hand, trace automation approaches reduce the effort of trace generation, but also produce a certain amount of false positives (wrong traces). These characteristics influence the correctness and completeness of traces, and the overall effectiveness of trace applications like change impact analysis.

Figure 44 gives an overview on available requirements tracing approaches and their influence on completeness and correctness of captured traces. Value-Based Requirements Tracing (VBRT) aims at tracing high-priority requirements with highest precision (higher precision results in a higher correctness). On the other hand VBRT does not provide complete trace sets, because low-priority requirements need not to be traced or only with low precision.

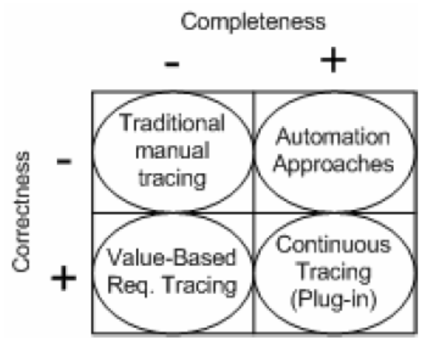


Figure 44 Requirements Tracing Approaches to improve completeness and correctness

Fully automate trace generation approaches tend to have a high degree of completeness; their disadvantage is a considerable number of false positives that hinder efficient change impact analysis. The most promising approach seems to be a combination of manual and automated tracing approaches. In general there are two possibilities how to combine these types of approaches: (a) automate trace generation and revise captured traces manually. An example for such an approach is heterogeneous requirements tracing described in [23]; (b) capture traces manually, but reduce efforts and complexity by tool support, e.g., with the plug-in based approach.

5.5.7 Validity of results

The purpose of the comparison presented in this paper was to investigate the impact of the continuous tracing approach on the efforts of tracing and the correctness and completeness of traces. In order to do that, I analyzed a class of projects with a medium to large size (150 to 300 requirements). For smaller projects, the tracing efforts might be too high compared to ad-hoc tracing; the investment for tracing seems in general questionable for projects with only a low number of requirements.

Furthermore, I only compared 3 alternatives which are, from our point of view, most relevant (mostly used) for practitioners. Other results might be achieved with other tools, other tracing processes or strategies.

5.5.8 Conclusion and Further Work

Requirements traceability is a good means for stakeholders to identify dependencies between artifacts, e.g., traces between a requirement and source code methods help developers to understand the source code and to identify change locations, due to a change request, more easily than without traces.

The challenge of requirements traceability is the considerably high effort to capture and maintain traces. Another problem is that stakeholders very often have to use extra tools for requirements management and tracing, e.g., a requirements management tools or particular tracing tools. As reported in literature, this often leads to a "yet-another-tool" syndrome and results in neglected requirements tracing tasks.

In this work I proposed a plug-in-based facilitation of requirements tracing. The plug-in based approach addresses the challenges above by supporting the following characteristics:

- Less cognitive complexity for developers to capture and maintain traces, because they work within their familiar work environment without using extra tools.
- Less delay of trace generation: developers can create and maintain traceability information easily during developing;
- More complete trace generation: By providing tracing facilities in their familiar work environment, the acceptance of developers to capture and maintain traces should increase.
- More correct trace generation: due to less delay, the captured trace information should be more correct than with an approach where code is developed first, and traces are captured later on.
- Less effort of RT: In comparison to other manual RT approaches, where cross-references or traceability matrices are created manually, the plug-in automates these tasks and reduces the effort for capturing trace information to a "double click".

I compared 3 tracing approaches to support change impact analysis. The results of our comparison were: a) tool support that allows capturing requirements within the development environment in less than 5 seconds had strong positive effects on the tracing acceptance by developers; and thus b) according to the effect model presented in Section 3 systematically supports more complete and correct traces and thereby improves the basis for a more realistic change impact analysis.

Further work will consist of a) the replication of empirical studies to provide a more comprehensive data set for the investigation of the effects modelled in Section 3 for other project contexts; b) a systematic comparison between improved manual "best practice" tracing approaches and "best in class" automatic trace generation approaches; and finally c) the extension of the integrated tool support concept beyond source code to other forms of tracing.

While I restricted the comparison to a comparison of efforts, correctness and completeness with other manual RT approaches, which are typically used in practice, further evaluation will also consist of a comparison with automated trace generation approaches like dynamic tracing.

Furthermore, factors depicted Figure 40, namely correctness and completeness of traces, and the influence on the effectiveness of change impact analysis need in-depth evaluation. I plan a more detailed evaluation with a controlled experiment as part of a "software engineering" workshop.

I further have to evaluate the scalability of the approach; e.g., if it is still feasible with increasing number of requirements. I expect the approach to have a good scalability, due to the following reasons: The import of the requirements list from Requisite Pro into the developers' work environment is not a bottleneck. It is more a matter of usability to suitably display a long list of requirements in the Eclipse plug-in. What the plug-in already provides is the possibility to select requirements types to be imported instead of importing all existing requirements. Furthermore, I want to improve scalability by adding filter mechanisms to the plug-in, which allows customizing the requirements' display in the plugin, so that only currently relevant requirements are displayed, e.g., scoping of relevant requirements for each development team.

The plug-in based concepts is currently developed also for other types of traces, e.g., for traces between requirements and test cases. At Siemens PSE, developers created an interface between Requisite Pro and SiTemppo (a test management tool). This interface allows importing requirements into SiTemppo, so test cases can be related to the requirements. The requirements and the traceability information can be re-imported into Requisite Pro where the traces between requirements and test cases are displayed.

Since the plug-in currently only exists for Eclipse and Requisite Pro, I also plan to build the similar plug-ins for Visual Studio and other requirements management tools like Doors, Caliber RM, and tools like Word and Excel.

Based on the promising results of the comparison and the effects model presented above, I am looking forward to systematically explore the relationships between the characteristics of trace generation approaches in order to find developer-friendly solutions that make high-quality tracing a normal part of software development and project management practice. Based on the results of the comparison, I expect the plug-inbased requirements tracing approach to have a significant influence on RT in software development projects.

5.6 Chapter Summary

Value-based Requirements Tracing supports project managers in planning tracing efforts according to requirements' priority: high-priority requirements are traced into source code at method level, with higher efforts; low-priority requirements are traced on package level (very cheap).

Initial results of the evaluation of our integrated developer tool support for dependency management are promising, because: the effort for capturing dependencies between requirements and source code elements is significantly reduced by providing semi-automated support for users and thereby improves trace quality such as correctness and completeness of captured traces.

6 APPLICATION OF TRACES FOR DEPENDENCY MANAGEMENT

Our contributions in chapter 4 and 5 focused on the management of technical dependencies, i.e. relationships between artefacts. In this chapter I focus on the usage of dependencies in a collaboration scenario of global software development (GSD) projects. In such a collaboration scenario project participants have to perform particular tasks, e.g. as usually defined in a project plan and they need context information in order to solve their tasks efficiently and effectively. This context information depends on the concrete role of the project participant, but generally consists of the following types of information:

- Historty of a task: are there preceding tasks to the current one and what are the results of these tasks
- Contact persons: who worked on preceding tasks and can provide helpful information
- Decisions that have been made and are relevant for the current task
- Related work products, e.g. a requirements engineer might need test results in order to have an overview on the status of a set of requirements.

Currently in GSD projects, such context information is usually available in separated tools, e.g., a requirements engineer uses a requirements management tool to specify a concrete requirement and can maintain additional requirements-related information in this tool. Or a tester manages the data and information necessary for his testing tasks in a test management tool.

The problem arises, when a tester needs additional information that cannot be retrieved from the test management tool, because it is stored in a different tool at another site or a project participant at another site could help. In such situations, the current approach to get this context information is to write an email (which can cause delay) or call the colleague at the other site (this option is limited by time-zone differences and the availability of colleagues at other sites)

Options that are often used to address these problems are:

- definition of work packages appropriate for distributed working (low coupling), e.g. putting packages with high communication effort together at one site, proper definition of interfaces and roles, and regular face-to-face meetings for discussions of interfaces between work packages (in case of major changes)
- Groupware tool support for organizational dependencies such as Netmeeting, Wikis, collaboration platforms with e-mail notification support so that relevant information of one site is automatically transported to other sites on certain events, e.g., document check-ins, etc.

Weaknesses of these existing approaches are that face-to-face meetings are expensive to establish and that current notification support generates many email events for changes (e.g. in Trac) and nobody is reading that due to low content information (consequence: people may take lot of effort for trivial things or choose to ignore or need to filter) Furthermore, there are weaknesses regarding user interface and customization capabilities of such tools.

I tackle the issues of currently available approaches mentioned above for managing social dependencies by providing the concept of a *role-based in-time notification system*. The idea of a role-based notification system is to complement the limited synchronous communication in GSD projects with asynchronous communication via the tool set used in a project that means that most of the information necessary for a project team member should be provided via the tools that he uses. Tool integration is again a prerequisite here in order to be able to retrieve relevant information from other tools in the project's tool set. Using traces in this context is a means to define which information is relevant and related to the current task of a project team member. By using this network of traces the project team members should be notified about events (changes) happening somewhere in the project that affect his current task.The number of notifications scales down to a degree where only really relevant notifications for a role in a particular situation are delivered. This role-based in-time notification system is based on the Application Lifecycle Management (tool integration) concept explained in section 2.4 and supports across-the-tools notifications.

The notification system approach is explained in the following section 6.1: subsection 6.1.1 describes the needs of GSD project participants as fundament for the notification system approach, and section 6.1.2 describes our concept.

6.1 Requirements Management Infrastructures in Global Software Development

The following subsection describes the need of GSD project members that I elicited at Siemens IT Solutions and Services. Then the concept for a trace-based notification system is derived.

6.1.1 Needs in GSD projects

In GSD projects typically multiple distributed teams work on the realization of requirements: e.g., project management (at site A), development teams (at sites B, C), and test teams (at site D) [67]. Typically these teams have many requirements-related tasks: project managers deal with the specification of requirements, change management, and requirements traceability; developers develop source code based on the provided set of requirements and report progress to project management (which requirements are already implemented?). Testers create test cases for requirements, execute test cases, and report tests, e.g., which test cases were completed successfully and thereby which requirements have been covered so far by testing Typically, these roles use tools that facilitate their work. A typical tool set in such projects may consist of a requirements management (REQM) tool, a configuration management tool, development tools (IDE), test management tools, collaboration platforms, and communication tools (like email and Netmeeting).

The Support Center Configuration Management (SCCM) frequently receives requests for REQM and configuration management (CM)-related support from GSD projects. During these consulting and support activities and in expert network meetings, where project participants from different projects come together to exchange and discuss current project-related REQM problems, the SCCM frequently receives feedback about REQM needs in GSD. Based on this feedback, I present the following list of main requirements management needs of PSE project participants in GSD projects:

Timely and effective information exchange (requirements awareness)

Notification about changes of relevant information, e.g., for change propagation as requirements are likely to change while developers work on their dependent implementation. As a consequence developers may implement outdated requirements, wasting effort on re-work of the code. Information about changes or other events at one site that are relevant for roles at other sites shall be provided timely, to avoid unnecessary rework. Furthermore, this information exchange shall be effective in the sense that each role gets exactly the information it needs (neither “flooded with email” nor “starving for information”), which is a challenge for project management in general but even more in GSD projects due to the limited availability of synchronous communication (geographical distribution across several time zones). Essential in that context is permanent accessibility of relevant information, e.g., a developer implementing a change request (CR), should have easy access to the task description tracing back to the original CR and, if necessary, further back to the documentation of related decisions and contact persons like stakeholders who may provide background information. It is important to maintain a clear picture of responsibilities in the project and dependencies (between artifacts, persons, and tasks) to allow the timely notification of relevant roles.

Damian and Zowghi [26] support the importance of such "requirements awareness mechanisms". Colocated teams usually benefit from social mechanisms and processes that facilitate the work practice and diminish the perceived need for explicit requirements awareness activities. However, this kind of access to informal communication is significantly limited in geographically distributed teams.

Requirements traceability

Tracing requirements [62] back to their origin or having rationale for them helps to better understand the meaning of requirements. In GSD projects traceability is even more important than in collocated projects, because requirements cannot be clarified easily during informal "corridor talks". Furthermore, captured traces between requirements and other artifacts like source code elements and test cases give the project manager hints for progress monitoring ("is requirement A already implemented and tested?"), change impact analysis ("which artifacts do I have to change when requirement B changes?"), or coverage analysis "(is every requirement sufficiently covered by test cases?").

Integration of tools

Comprehensive tool support is needed to enable consistent, error-free, and up-to-date information exchange, requirements awareness, and traceability in a GSD context. Tool support mostly consists of tool sets (requirements, development, and test tools) that can interact in principle providing the basis for redundancy-free, consistent storage of data and exchange of data between tools (via tool interfaces).

However, tighter tool integration than provided is needed, e.g., for project managers who want to see certain parts of test data from within the requirements management tool, so they do not have to use the test management tool itself.

In summary, tool support for requirements management in GSD projects has to meet the following success criteria: (1) permanent access to relevant information, e.g., history of a requirement; who worked on which requirement when, which decisions were made why (requirements awareness); (2) timely notification of relevant role on occurrence of specified events and conditions, e.g., changes to some requirements or dependent documents; (3) easy means to facilitate requirements tracing across tool borders as a prerequisite to manage dependencies; and (4) the right, i.e., higher than currently available, level of tool integration.

6.1.2 A Concept for a Role-Based Notification System

In order to fully address the REQM needs GSD project participants need a generic mechanism to ensure timely notifications about changes to requirements or other artifacts in addition to the common data exchange format mentioned above. Multiple requirementsrelated events may happen concurrently at several sites across a GSD project (see overview in Table 1), which would seem relevant for project team members at other sites. For example, when requirement 4711 in the requirements management tool changes (event in the requirements management tool), the developer who works on this requirement can be notified, e.g., by displaying and highlighting the requirement in the IDE (notification).

Another scenario is build automation: there is a tester that monitors test execution. If a test run fails, the project manager, who needs information about test progress, and the responsible development team, which will have to correct the bugs, need to be informed. Instead of forcing the tester to actively inform the project manager and developers by telephone or mail, it would be much cheaper if the tool infrastructure notifies them automatically, e.g., send the project manager a notification within his requirements management tool and let him view test results there, too. The developers get a notification with in their IDE and can view failed test cases in order to correct the relevant code.

Notifications are currently often triggered by persons, e.g., via telephone calls or emails. As explained above, this way of notification is often costly and unreliable; effective tool-based notifications promise to reduce the communication effort. Of course, the concept of notification is not new, e.g., there are already database triggers that implement some form of notifications. A novelty is the integration of events coming from heterogeneous systems as input to a rule engine that can correlate, aggregate, and filter events in order to provide triggers for relevant role-oriented notifications.

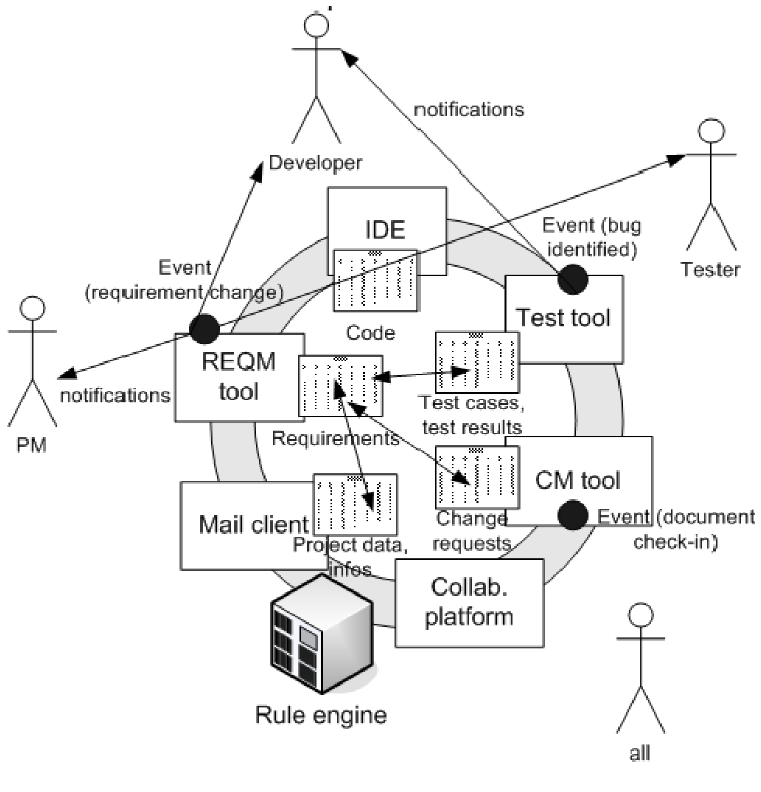


Figure 45 Role-Oriented Notification System

Based on the implementation of a common data exchange format, which can form a tool integration bus, I develop a notification system that can assist project users in defining useful notifications across GSD sites.

The notification system should be role-based, so that a user can define which kinds of notifications are relevant for him. As main challenges I see [65]:

- Provide useful notifications (correct, helpful; e.g., receiver would be willing to pay for getting a certain kind of message);
- Avoid information overload with irrelevant or wrong notifications;
- Keep an overview on the correctness of a large rule set; e.g., which rules should be active.

I want to implement such a notification by means of a tool-based rule engine, e.g., a correlated events proc-essor (CEP), which is connected to the set of used tools in project via a common “tool integration bus”. Figure 1 illustrates how the used tools are con-nected via that bus. The rule engine records events from the other tools and generates notifications for relevant roles/users via the used tools (eventually an ontology will be necessary to implement that, because of the different role sets in the used tools). Thereby, efforts for notifying other users are shifted from the users to the tool infrastructure, so that affected roles are informed in a timely manner.

Furthermore, a role-based notification system should provide three types of information: (1) artifact informa-tion: accurate information about the current state of a document; (2) event information: when an event occurs (e.g., change to a document) relevant roles can be noti-fied by the system; (3) artifact history: events are stored in a database and related to artifacts so that users can search the database for events that happened to re-quirements and other documents in the past.

Table 27 Examples for events in GSD project tools

Events in the REQM tool
new requirement is inserted
Existing requirement is changed
New user is added
User privileges are changed
Events in the IDE
Trace is created between source code element and requirement
Events in the Configuration Management tool
New change request was submitted
State of an artifact changed
Check-in of an artifact
Events in the test management tool
Bug report is inserted
Test cases for a particular requirement are reported as successfully tested.

I plan the following next steps to implement such a role- and event-based notification system:

1. Definition of notification rules

I need to have a syntax to formulate rules (notifica-tion rules) that determine notifications: Notification rules describe <whom> to notify <in what way> (e.g., e-mail, SMS, entry in change log)

<when> (e.g., immediately; batch every hour/day) due to <what change>.

Whom: can be a list of persons or roles.

Change can be an observable event or state change regarding an artifact or project state, e.g., some expected event did not happen during the given time win-dow (see further examples in Table 27). An SQL-like example rule for the build automation scenario in section 3 could be:

if TESTRUN 0815 FAILS NOTIFY (PM, Pe-ter Mayer) BY (ReqPro, IDE) and SEND FAILED TEST CASES

The rule says that in case of an erroneous test run the project manager (who ever has this role) should be in-formed by getting a notification in Requisite Pro, and Peter Mayer (a developer) gets notified by his IDE. The "send" option defined which data should be con-tained in the notification, e.g., the failed test cases as info for the PM and the developers, so that they know, which code to check.

2. Escalation

If a condition in the rule set indicates a state that the system cannot handle, the default is to escalate the is-sue with appropriate context information to a role that is expected to have enough overview and competence to provide a reasonable decision.

3. Tool support

Tools can support role-oriented notification by inter-pretation of machine-readable dependency information (that would be cumbersome for humans to follow). In a next step I want to use CEP to enable notifica-tions. CEP correlates related events to efficiently filter interesting events as basis for notification. Recent noti-fications can be stored in a database, so users can choose whether to

- receive change notifications immediately or sum-marized in regular time intervals;
- look at recent changes in the change database;
- look at the current version of artifacts (if there are many changes and/or major structural changes).

The CEP receives events from the tools, processes them according to a rule set and sends notifications to users via the tools. The processing of the events is based on user-defined notification rules. Each user can define for which events he wants to be notified. The challenge is to configure the CEP so that users get the relevant information they need in a timely manner, but on the other hand are not spammed with unimportant notifications. Value-based principles [10] will be applied here.

7 CONCLUSION AND FURTHER WORK

Global software development (GSD) projects are complex due to the high number of requirements, global distribution of project participants, and high number of dependencies between artefacts (e.g., relationships between requirements, or between requirements and test cases).

A key challenge in GSD projects is to cope with requirement and artifact changes that occur concurrently along the life cycle. Often changes done only punctiform in certain artefacts threaten the consistency among artefacts. Managing dependencies is crucial for implementing changes consistently, but current dependency management approaches are too expensive or, in case of automated dependency management approaches, provide too low quality of captured dependencies (i.e. wrong dependencies and/or incomplete sets of dependencies).

In this work, I proposed research contributions that significantly improved the cost-benefit of available dependency management approaches. The research contributions fit into the following areas: 1) planning dependency management, 2) capturing dependencies explicitly, systematically, and consistently, and 3) application of explicit dependencies (traces).

Research Issue: Planning of dependency management – Tracing Activity Framework

Managing all dependencies that exist in GSD projects would cause considerable high efforts and tool challenges (which dependencies should be maintained with which tools?). Thus, I proposed a tracing activity framework (TAF) that supports a project manager in (a) defining which dependencies should be managed and (b) comparing dependency management approaches in advance (before dependency management starts in the project) to find the most cost-effective approach. TAF consists of tracing activities and events like trace generation, deterioration, validation, rework, and trace usage. For each tracing activity, the relevant parameters that influence the activity are assigned.

I performed a case study to evaluate the feasibility of the TAF and to model 3 different tracing strategies, namely no tracing, systematic full tracing, and value-based tracing for re-testing support; in our case with focus on effort, also considering delay, and risk of tracing strategies.

Modelling the 3 tracing strategies by using TAF activities and parameters provided data points for effort of each strategy. As these are single data points in a specific study setting, I see the results as snap shots, which should motivate further data collection to allow statistical data analysis and sensitivity analysis. As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings.

Nevertheless, the tracing activity model was found useful for systematically modeling and evaluating the tracing alternatives. The model helps to make alternative strategies comparable, as it makes the main tracing activities explicit and allows mapping of relevant parameters that influence tracing costs and benefits. According to the expert feedback the calculated efforts provide a good input to reasoning about which tracing strategy seems most beneficial in a particular project context.

Research Issue: Capturing dependencies explicitly, systematically, and consistently

Capturing dependencies explicitly (e.g. by requirements tracing matrices or tables) is helpful for change impact analysis.

In this work I proposed to approaches that support the explicit, systematic, and consistent capture of dependencies in a cost-effective way: (a) value-based requirements tracing, and (b) an integrated developer tool support for requirements tracing.

Value-Based requirements tracing

Existing methods, tools and approaches of requirements tracing in literature do not take economic issues of dependency management into consideration: due to effort and budget constraints it is often impossible to trace each requirement at highest level of detail. Value-based approaches allow tailoring efforts according to requirements' priority. In context with requirements traceability, I interpreted this as to use trace types of variable precision. For example, I used three different trace types to trace the requirements into source code, namely method traces, class traces, and package traces. The first trace type allows tracing requirements into code at method level, the second trace type allows tracing requirements into code at class level, and the third trace type allows tracing requirements into code at package level. This reflects a level of precision and also effort necessary to create these traces, e.g., tracing into methods is more expensive than tracing into classes.

I evaluated a value-based requirements tracing (VBRT) approach in a real-life project setting and compared costs and benefits of VBRT with ad hoc tracing and full tracing. Main results of the case study were: *(a) VBRT took around 35% effort compared to full tracing; (b) more risky requirements need more detailed tracing. The case study results illustrate that VBRT is an attractive tracing alternative for typical software development projects in comparison with ad hoc tracing and full tracing, because it provides "traditional" benefits of tracing and thereby minimizes tracing efforts.*

Requirements tracing is important to keep track of the dependencies between requirements and other artifacts and to support project teams and software maintenance personnel in several tasks, e.g. change impact analysis, requirements conflict identification, consistency checking. VBRT is a promising approach to alleviate the problem of high effort of requirements tracing in a practical and comprehensible way.

Integrated developer tool support

The challenge of requirements traceability is the considerably high effort to capture and maintain traces. Another problem is that stakeholders very often have to use extra tools for requirements management and tracing, e.g., a requirements management tools or particular tracing tools. As reported in literature, this often leads to a "yet-another-tool" syndrome and results in neglected requirements tracing tasks.

In this work I proposed a plug-in-based facilitation of requirements tracing. The plug-in based approach addresses the challenges above by supporting the following characteristics:

- Less cognitive complexity for developers to capture and maintain traces, because they work within their familiar work environment without using extra tools.
- Less delay of trace generation: developers can create and maintain traceability information easily during developing;
- More complete trace generation: By providing tracing facilities in their familiar work environment, the acceptance of developers to capture and maintain traces should increase.
- More correct trace generation: due to less delay, the captured trace information should be more correct than with an approach where code is developed first, and traces are captured later on.
- Less effort of RT: In comparison to other manual RT approaches, where cross-references or traceability matrices are created manually, the plug-in automates these tasks and reduces the effort for capturing trace information to a "double click".

I compared 3 tracing approaches to support change impact analysis. The results of the comparison were: a) tool support that allows capturing requirements within the development environment in *less than 5 seconds had strong positive effects on the tracing acceptance by developers; and thus b) systematically supports more complete and correct traces and thereby improves the basis for a more realistic change impact analysis.*

Research Issue: Application of explicit dependencies (traces)

Currently in GSD projects, context information necessary for performing a particular task is usually spread out over separated tools, e.g., a project manager uses a requirements management tool to specify a concrete requirement and can maintain additional requirements-related information in this tool. Or a tester manages the data and information necessary for his testing tasks in a test management tool.

The problem arises, when a tester needs additional information that cannot be retrieved from the test management tool, because it is stored in a different tool at another site or a project participant at another site could help. In such situations, the current approach to get this context information is to write an email (which can cause delay) or call the colleague at the other site (this option is limited by time-zone differences and the availability of colleagues at other sites)

I tackled the issues of currently available approaches mentioned above by providing the concept of a *role-based in-time notification system*. The idea of a role-based notification system is to complement the limited synchronous communication in GSD projects with asynchronous communication via the tool set used in a project. Tool integration is a prerequisite here in order to be able to retrieve relevant information from other tools in the project's tool set. Using traces in this context is a means to define which information is relevant and related to the current task of a project team member. By using this network of traces the project team members should be notified about events (changes) happening somewhere in the project that affect his current task.The number of notifications scales down to a degree where only really relevant notifications for a role in a particular situation are delivered. Further work will consist of building a prototype for a role-based notification system (see further work).

In summary, the lessons learned of my overall research work are:

- *Dependency management is the basis for project management and needs to be planned.*
- *A process model (Tracing Activity Model) for dependency management (DM) and tracing was found useful to structure research and evaluate tracing alternatives.*
- *Value-based DM (focus on high benefit/risk dependencies) is useful to provide high-value dependency information by using the usually very limited resources and budget.*
- *Integrated tool support for tracing was very useful for developers because they could easily manage dependency information from within their usual tool (development environment) in a very unexpensive way.*
- *Role-based notification was found interesting to provide relevant context information for project participants across tool borders and sites, but needs more research in a variety of project contexts for better external validity.*

The purpose of the case studies presented in this paper was to investigate the impact of our contributions in the field of dependency management regarding effort reduction, trace quality, and cost-benefit of requirements tracing. Thus the case study sizes were chosen to allow comparing particular approaches in a reasonable amount of time. However, the case study project settings were typical in the given companies and allow reasonable insight into the feasibility of our dependency management approaches in this environment. I see the empirical investigations as initial studies that support planning of further empirical studies with larger projects. As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings. My research contributions significantly improved the cost-benefit of available dependency management approaches.

7.1 Further Work

Further work will mainly focus on (a) replication of case studies to confirm the findings for value-based requirements tracing and the integrated developer tool support, (b) building and evaluate a prototype for a trace-based in-time notification system based on the concept I have outlined in section 6, and (c) improving methods for capturing dependencies, e.g. by investigating new technological concepts like ontologies.

7.1.1 Replication of Case Studies to Confirm the Findings for Value-Based Requirements Tracing and Integrated Developer Tool Support

Further work will consist of a) the replication of empirical studies to provide a more comprehensive data set for the investigation of the effects modelled in Section 3 for other project contexts; b) a systematic comparison between improved manual "best practice" tracing approaches and "best in class" automatic trace generation approaches; and finally c) the extension of the integrated tool support concept beyond source code to other forms of tracing.

While I restricted our comparison to a comparison of efforts, correctness and completeness with other manual RT approaches, which are typically used in practice, further evaluation will also consist of a comparison with automated trace generation approaches like dynamic tracing.

Furthermore, factors depicted in Figure 40, namely correctness and completeness of traces, and the influence on the effectiveness of change impact analysis need in-depth evaluation. I plan a more detailed evaluation with a controlled experiment as part of a "software engineering" workshop.

I further have to evaluate the scalability of the integrated tracing approach; e.g., if it is still feasible with increasing number of requirements. I expect the approach to have a good scalability, due to the following reasons: The import of the requirements list from a requirements management tool into the developers' work environment is not a bottleneck. It is more a matter of usability to suitably display a long list of requirements in the Eclipse plug-in. What the plug-in already provides is the possibility to select requirements types to be imported instead of importing all existing requirements. Furthermore, I want to improve scalability by adding filter mechanisms to the plug-in, which allows customizing the requirements' display in the plugin, so that only currently relevant requirements are displayed, e.g., scoping of relevant requirements for each development team.

The plug-in based concepts is currently developed also for other types of traces, e.g., for traces between requirements and test cases. At Siemens PSE, developers created an interface between Requisite Pro and SiTemppo (a test management tool). This interface allows importing requirements into SiTemppo, so test cases can be related to the requirements. The requirements and the traceability information can be re-imported into Requisite Pro where the traces between requirements and test cases are displayed.

Since the plug-in currently only exists for Eclipse and Requisite Pro, I also plan to build the similar plug-ins for Visual Studio and other requirements management tools like Doors, Caliber RM, and tools like Word and Excel.

I am looking forward to systematically explore the relationships between the characteristics of trace generation approaches in order to find developer-friendly solutions that make high-quality tracing a normal part of software development and project management practice. Based on the results of the comparison, I expect the plug-in-based requirements tracing approach to have a significant influence on RT in software development projects.

7.1.2 Development and Evaluation of Ontology-Based Tool Integration Approaches to Support Dependency Management and Analysis

As mentioned above, different artefacts of a project are typically stored and maintained in different tools, e.g. requirements management tools, test management tools, development environments, configuration management repositories.

Project participants need tool integrations to maintain dependencies between these artefacts, e.g. to relate a set of test cases to a particular requirement (to highlight that these test cases cover the requirement).

The typical practice to integrate such tools is to build point-to-point integrations between particular tools, as depicted in the left part of Figure 46. This makes it rather expensive to integrate tools with each other. Furthermore these integrations are often limited in flexibility, i.e. data that can be exported / imported between tools.

A promising approach to solve these issues is to use ontologies for tool integrations, as depicted on the right part of Figure 46.

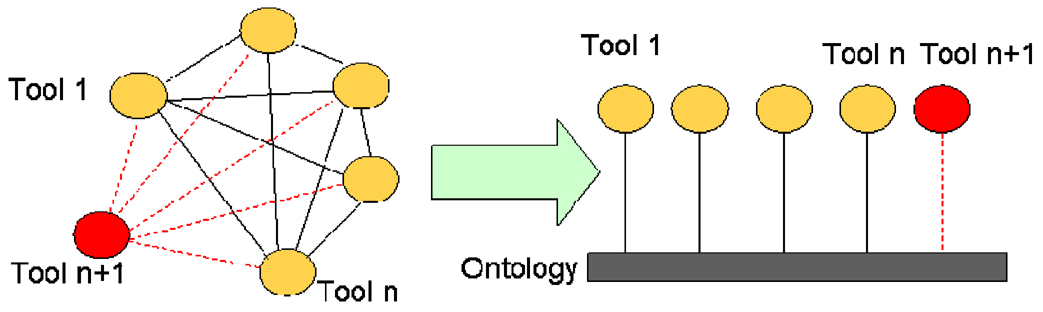


Figure 46 Ontologies as Means for Tool Integrations

An ontology is a representation that allows both to display a conceptual model, e.g. containing types of requirements, test cases and the types of dependencies between these elements, and concrete instances (data records) of these elements. Ontology representations are very flexible and thus seem to be a good means for tool integrations in a sense that all data types and data records from different tools can be imported into an ontology, dependencies between all kinds of elements can flexibly be created and analysed there, and finally the captured dependency information can be re-imported into particular tools and displayed there appropriately.

The benefits of using ontologies for tool integrations are:

- It is an external uniform database for all kinds of tools;
- Uniform manipulation of data with ontology reasoning for dependency analysis and transformation of data.
- Even if tools allow only configuration of data fields and user interface, the ontology database can provide the functionality externally and exchange the relevant data with the tools as needed.
  - Checks for consistency and plausibility
  - Transformation for mapping dependencies

On the other hand, there might also be some limitations for ontologies: Ontologies may be slower than a relational database, which may limit the size of data to manipulate, which raises the need to focus on most worthwhile data.

I will build and evaluate a prototype that allows (a) export of data from different tools into one common ontology, (b) capture of dependencies and dependency analysis in the ontology, and (c) re-import of dependency and analysis data into other tools, so that particular roles in the project can retrieve relevant data from their particular tool.

7.1.3 Development and Evaluation of an Notification System Prototype

I have mentioned above the necessity of tool integration for notifications across tool borders. In a first step I want to implement our notification concept in a prototype by using a lightweight ticket management system such as Trac (in combination with the configuration management Subversion).

The advantages of using Trac for a role-based in-time notification system are:

- Trac (as well as Subversion) is open source software, which makes it cheap to get and easy to adapt/extend
- Trac is a lightweight (easy-to-use because of simple concepts) ticket management tool that gets more and more popular in industry
- Trac is a ticket management tool. Thus, every project participant typically uses such a tool by default (in contrast to particular CASE tools like requirements or test management tools). This is a good prerequisite for providing a project-wide notification system for all existing project participants.
- Trace has a basic notification mechanism built in (e-mail notification), no notification rules are implemented, but can be extended

After building this notification system prototype I want to evaluate it in order to (a) analyse, how the notification rules can be supported by the prototype and how value-based (role-based) approaches for rule defintions worked, (b) get feedback of real project members about the user-defined notification rules, and (c) identify further technical challenges and options for improvement.

The future work mentioned above will further optimize the way of capturing dependencies and increase the value of traces by exploring new utilization areas, e.g. in a role-based notification system.

8 REFERENCES

[1] Alford, M. (1977), 'A Requirements Engineering Methodology for Real-Time Processing Requirements', IEEE Trans. on Software Engineering Vol. SE-3, No. l., 60-69.

[2] Carina Alves, Anthony Finkelstein, Workshop on software engineering decision support: components: Challenges in COTS decision-making: a goal-driven requirements engineering perspective, Proc. 14th int. conf. on Software engineering and knowledge engineering, SEKE '02, July 2002

[3] Anton, A.; Carter, R.; Dagnino, A.; Dempster, J. & Siege, D. (2001), 'Deriving Goals from a Use-Case Based Requirements Specification', Requirements Engineering Journal 6.

[4] Antoniol, G.; Canfora, G.; Casazza, G.; Lucia, A. D. & Merlo, E. (2002), 'Recovering Traceability Links between Code and Documentation', Transactions on Software Engineering, 970-983.

[5] Antoniol, G.; Caprile, B.; Potrich, A. & Tonella, P. (2001), 'Design-Code Traceability Recovery: Selecting the Basic Linkage Properties', Science of Computer Programming vol. 40, no. 2-3, pp. 213-234.

[6] Archer, J. (2003), 'Requirements Tracking: A Streamlined Approach''IEEE International Conference on Requirements Engineering'.

[7] Arkley, P.; Manson, P. & Riddle, S. (2002), Position Paper: Enabling Traceability, in '1st International Workshop on Traceability in Emerging Forms'.

[8] Arkley, P. & Riddle, S. (2005), 'Overcoming the Traceability Benefit Problem''IEEE International Conference on Requirements Engineering'.

[9] Arkley, P.; Riddle, S. & Brookes, T. (2006), 'Tailoring Traceability Information to Business Needs''IEEE International Conference on Requirements Engineering'.

[10] Biffl, S.; A., A.; B., B.; H., E. & P., G. (2005), Value-based Software Engineering, Springer.

[11] Biolchini, J.; Mian, P.; Natali, A. & Travasso, G. (2005), 'Systematic Review in Software Engineering', Technical report, Relatyrio Tйcnico ES-679/05, PESC-COPPE/UFRJ.

[12] Boehm, B. (2003), 'Value-Based Software Engineering', ACM SIGSOFT Software Engineering Notes vol 28 no 2.

[13] Boehm, B. & TurnerWesley, A., ed. (2005), Balancing Agility and Discipline,, Addison Wesley.

[14] P. Bourque, R. Dupuis, A Abran, The Guide to the Software Engineerin Body of Knowledge, IEEE Software, Nov/Dec. 1999

[15] Brereton, P.; Kitchenham, B.; Budgen, D.; Turner, D. & Khalil, M. (), 'Employing Systematic Literature Review: An Experience Report', 2005.

[16] Brereton, P.; Kitchenham, B.; Budgen, D.; Turner, M. & Khalil, M. (2007), 'Lessons from applying the systematic literature review process within the software engineering domain', The Journal of Systems & Software.

[17] Chisan, J. & Damian, D. (2005), 'Exploring the role of requirements engineering in improving risk management''IEEE International Conference on Requirements Engineering'.

[18] Cleland-Huang, J. (2006), 'Requirements Traceability - When and How does it Deliver more than it Costs?"International Conference on Requirements Engineering'.

[19] Cleland-Huang, J.; Chang, C. K. & Christensen, M. J. (2003), 'Event-Based Traceability for Managing Evolutionary Change', Transactions on Software Engineering, 796-810.

[20] Cleland-Huang, J.; Chang, C. K.; Sethi, G.; Javvaji, K.; Hu, H. & Xia, J. (2002), 'Automating Speculative Queries through Event-Based Requirements Traceability"International Conference on Requirements Engineering'.

[21] Cleland-Huang, J.; Settimi, R.; Duan, C. & Zou, X. (2005), 'Utilizing, Supporting Evidence to Improve Dynamic Requirements Traceability"IEEE International Conference on Requirements Engineering'.

[22] Cleland-Huang, J.; Settimi, R.; Khadra, O. B.; Berezhanskaya, E. & Christina, S. (2005), 'Goal-centric traceability for managing non-functional requirements"International Conference on Software Engineering'.

[23] Cleland-Huang, J.; Zemont, G. & Lukasik, W. (2004), 'A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability"IEEE International Conference on Requirements Engineering'.

[24] Cochrane, 'Cochrane Collaboration, http://www.cochrane.org, accessed May 14 2007'.

[25] Costello, R. J. & Liu, D. (1995), 'Metrics for requirements engineering', Journal of Systems and Software.

[26] Damian, D. & Zowghi, D. (2003), 'Requirements Engineering challenges in multi-site software development organizations', Requirements Engineering Journal 8, 149-160.

[27] Dashofy, E.; Hoek, A. & Taylor, R. N. (2001), 'A Highly-Extensible, XML-Based Architecture Description Language"Working IEEE/IFIP Conference on Software Architectures'.

[28] Davis, A. (1990), 'The analysis and specification of systems and software requirements', Systems and Software Requirements Engineering, IEEE Computer Society Press, 119-144.

[29] Davis, A. M.; Tubío, Ó. D.; Hickey, A. M.; Juzgado, N. & Moreno, A. M. (2006), 'Effectiveness of Requirements Elicitation Techniques: Empirical Results Derived from a Systematic Review"International Conference on Requirements Engineering'.

[30] Department of Defense, U. (1998), 'Military standard: Defense systems software development', DOD-STD-2167A.

[31] Dick, J. (2005), 'Design Traceability', IEEE Software 22.

[32] Dömges, R. & Pohl, K. (1998), 'Adapting Traceability Environments to Project-Specific Needs', Communications of the ACM Vol. 41, No. 12, 54-62.

[33] Ebner, G. & Kaindl, H. (2002), 'Tracing All Around in Reengineering', IEEE Software 19.

[34] Egyed, A.Verlag, S., ed. (2005), Tailoring Software Traceability to Value-based Needs, In: Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., and Grünbacher, P. (eds.), Value-Based Software Engineering Springer Verlag.

[35] Egyed, A. (2001), 'A Scenario-Driven Approach to Traceability''International Conference on SoftwareEngineering'.

[36] Egyed, A.; Biffl, S.; Heindl, M. & Grünbacher, P. (2005), 'Determining the cost-quality trade-off for automated software traceability''IEEE/ACM International Conference on Automated Software Engineering'.

[37] Egyed, A. & Grünbacher, P. (2005), 'Supporting Software Under-standing with Automated Traceability', International Journal of Software Engineering and Knowledge Engineer-ing (IJSEKE) (in press).

[38] Egyed, A. & Grünbacher, P. (2004), 'Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help', IEEE Software 8.

[39] Egyed, A. & Grünbacher, P. (2003), 'A Scenario-Driven Approach to Trace Dependency Analysis', Transactions on software engineering, 116-132.

[40] Egyed, A. & Grünbacher, P. (2002), 'Automating Requirements Traceability: Beyond the Record & Replay Paradigm''IEEE/ACM International Conference on Automated Software Engineering'.

[41] Egyed, A.; Grünbacher, P.; Heindl, M. & Biffl, S. (2007), 'Value-Based Requirements Traceability: Lessons Learned''15th IEEE International Requirements Engineering Conference (RE'07), India Habitat Center, New Delhi, October 15-19th'.

[42] Elbaum, S.; Gable, D. & Rothermel, G. (2001), 'Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites''IEEE METRICS'.

[43] IEEE Std 830-1993, 'Recommended Practice for Software Requirements Specifications (December 2, 1993)'.

[44] Evans, M.Sons, J. W. &., ed. (1989), The Software Factory, John Wiley & Sons

[45] Felici, M. (2004), 'Observational Models of Requirements Evolution', Technical report, University of Edinburg.

[46] Frakes, W. & Baeza-Yates, R.Cliffs, E., ed. (1992), Information Retrieval: Data Structures and Algorithms, N.J.: Prentice-Hall.

[47] Frankl, P. G.; Rothermel, G. & Sayre, K. (2003), 'An Empirical Comparison of Two Safe Regression Test Selection Techniques''Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)'.

[48] Freimut, B., Punter, T., Biffl, St., Ciolkowski, M., ‚State of the art in empirical studies`, Technical Report, Virtuelles Software Kompetenzzentrum, 2002

[49] Gates, A. Q. & Mondragon, O. (2002), 'FasTLInC: a constraint-based tracing approach', Journal of Systems and Software 7.

[50] Gotel, O. & Finkelstein, A. (1997), 'Extended Requirements Traceability: Results of an Industrial Case Study''IEEE International Conference on Requirements Engineering'.

[51] Gotel, O. & Finkelstein, A. (1994), 'An analysis of the requirements traceability problem''IEEE International Conference on Requirements Engineering'.

[52] Gottesdiener, E.; Requirements by Collaboration – Workshops for Defining Needs; Addison-Wesley, 2002

[53] Gross, D. & Yu, E. (2001), 'From Non-Functional Requirements to Design through

Patterns', Engineering Journal 72, 18-36.

[54] Harker, S. & Eason, K. (1992), 'The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering''IEEE'.

[55] Haumer, P.; Heymans, P.; Jarke, M. & Pohl, K. (1999), 'Bridging the gap between past and future in RE: a scenariobased approach''IEEE International Conference on Requirements Engineering'.

[56] Hayes, J. H.; Dekhtyar, A. & K.Sundaram, S. (2006), 'Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods', Transactions on Software Engineering.

[57] Hayes, J. H.; Dekhtyar, A. & Osborne, J. (2003), 'Improving Requirements Tracing via Information Retrieval''IEEE International Conference on Requirements Engineering'.

[58] Hayes, J. H.; Dekhtyar, A.; Sundaram, S. K. & Howard, S. (2004), 'Helping Analysts Trace Requirements: An Objective Look''IEEE International Conference on Requirements Engineering'.

[59] Heindl, M. & Biffl, S. (2007), 'Requirements Tracing Strategies for Change Impact Analysis and Re-Testing - An Initial Tracing Activity Model and Industry Feasibility Study', Technical report, Vienna University of Technology.

[60] Heindl, M. & Biffl, S. (2007), 'A Framework to Balance Tracing Agility and Formalism''CEE-SET'.

[61] Heindl, M. & Biffl, S. (2006), 'The Impact of Trace Correctness Assumptions on the Cost-Benefit of Requirements Tracing''5th ACM/IEEE International Symposium on Empirical Software Engineering 2006 (ISESE 2006)'.

[62] Heindl, M. & Biffl, S. (2006), 'Project management: Risk management with enhanced tracing of requirements rationale in highly distributed projects''Proceedings of the 2006 international workshop on Global software development for the practitioner GSD '06'.

[63] Heindl, M. & Biffl, S. (2005), 'A case study on value based requirements tracing''10th European software engineering conference'.

[64] Heindl, M.; Biffl, S.; Grünbacher, P. & Egyed, A. (2005), 'A Value-Based Approach for Understanding Cost-Benefit Trade-Offs During Automated Software Traceability''Workshop on Tracebility in Emerging Forms of Software Engineering (TEFSE)'.

[65] Heindl, M.; Reinisch, F. & Biffl, S. (2007), 'Requirements Management Infrastructures in Global Software Development - Towards Application Lifecycle Management with Role-Oriented In-Time Notification''International Conference on Global Software Engineering - REMIDI workshop'.

[66] Heindl, M.; Reinisch, F.; Biffl, S. & Egyed, A. (2006), 'Value-Based Selection of Requirements Engineering Tool Support''32nd EUROMICRO Conference on Software Engineering and Advanced Applications 29-01 Aug. 2006 Page(s):266 - 273'.

[67] Herbsleb, J. D.; Paulish, J. & Bass, M. (2005), 'Global Software Development at Siemens: Experiences from Nine Projects''International Conference of Software Engineering (ICSE)'.

[68] Hoffmann, M.; Kuhn, N.; Weber, M.; Bittner, M.; Requirements for requirements

management tools, Requirements Engineering Conference, 2004. Proc. 12th IEEE International, 2004 Page(s):301 – 308

[69] Hsia, P.; Gao, J.; Samuel, J.; Kung, D.; Toyoshima, Y. & Chen, C. (1994), 'Behavior-based Acceptance Testing of Software Systems: A Formal Scenario Approach', IEEE Computer.

[70] Huffman-Hayes, J.; Dekhtyar, A. & Sundaram, S. (22), 'Improving after-the-fact tracing and mapping: supporting software quality predictions', IEEE Software 2005.

[71] International Council on Systems Engineering, http://www.paper-review.com/tools/rms/read.php

[72] Jackson, J. (1991), 'A Keyphrase Based Traceability Scheme''IEE Colloquium on Tools and Techniques for Maintaining Traceability during Design'.

[73] Jarke, M. (1998), 'Requirements Traceability', Communications of the ACM Vol. 41, No. 12.

[74] Jedlitschka, A. & Pfahl, D. (2003), 'Experience-Based Model-Driven Improvement Management with Combined Data Sources from Industry and Academia''ISESE'.

[75] Juristo, N.; Moreno, A. M. & Vegas, S. (2004), 'Reviewing 25 Years of Testing Technique Experiments', Journal Empirical Software Engineering Volume 9, 7-44.

[76] Kaindl, H. (1997), 'A Practical Approach to Combining Requirements Definition and Object-Oriented Analysis', Annals Software Eng. vol. 3, 319–343.

[77] Kaindl, H. (1993), 'The Missing Link in Requirements Engineering', ACM SigSoft Soft. Eng. Notes vol. 18, no. 2, pp. 30-39.

[78] Karlsson, J. (1996), 'Software Requirements Prioritizing''Proceedingsof the 2nd International Conference on Requirements Engineering (ICRE'96), Colorado Springs,Colorado, April 15-18'.

[79] Kitchenham, B. (2004), 'Procedures for Performing Systematic Reviews', Technical report, Keele University, UK.

[80] Kitchenham, B. A.; Pfleeger, S. L.; Pickard, L. M.; Jones, P. W.; Hoaglin, D. C.; Emam, K. E. & Rosenberg, J. (2002), 'Preliminary Guidelines for Empirical Research in Software Engineering', IEEE Transactions on Software Engineering, vol. 28, no. 8.

[81] Knethen, A. .V. & Paech, B. (2002), 'A Survey on Tracing Approaches in Practice and Research''IESE-Report No. 095.01/E unpublished'.

[82] Knethen, A. V.; Paech, B.; Kiedaisch, F. & Houdek, F. (2002), 'Systematic Requirements Recycling through Abstraction and Traceability''International Conference on Requirements Engineering'.

[83] Kotonya, G. & Sommerville, I.Wiley, ed. (1997), Requirements Engineering – Processes and Techniques, Wiley.

[84] Laitenberger, O. & DeBaud, J. (2000), 'An encompassing life cycle centric survey of software inspection', An encompassing life cycle centric survey of software inspection.

[85] Lang, M. & Duggan, J. (2001), 'A Tool to Support Collaborative Software Requirements Management', Requirements Engineering Journal 6.

[86] Lange, D. & Nakamura, Y. (1997), 'Object-oriented program tracing and visualiza-

tion', IEEE Computer 30.

[87] Louis A. Le Blanc, Willard M. Korn, A structured approach to the evaluation and selection of CASE tools, Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing, March 1992

[88] Lefering, M. (1993), 'An Incremental Integration Tool between Requirements Engineering and Programming in the Large''Proc. IEEE International Symp. on Requirements Engineering,San Diego, California'.

[89] Lin, J.; Lin, C. C.; Cleland-Huang, J. & Settimi, R. (2006), 'Poirot: A Distributed Tool Supporting Enterprise-Wide Automated Traceability''IEEE International Conference on Requirements Engineering'.

[90] Macfarlane, I. & Reilly, I. (1995), 'Requirements Traceability in an Integrated Development Environment''International Conference on Requirements Engineering'.

[91] Neil A. M. Maiden, Cornelius Ncube, Andrew Moor, Lessons learned during requirements acquisition for COTS systems, Communications of the ACM, Vol.40, Issue 12, Dec. 1997

[92] Marcus, A. & Maletic, J. (2003), 'Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing''International Conference on Software Engineering'.

[93] Mendes, E. (2005), 'A systematic review of Web engineering research''ISESE'.

[94] Mohagheghi, P. (), 'Global Software Development: Issues, Solutions, Challenges', Dept. Computer and Information Science (IDI)University of Science and Technology (NTNU)Trondheim, Norwayparastoo@idi.ntnu.noTrial lecture, 21 September 2004.

[95] Murta, L.; van der Hoek, A. & Werner, C. (2006), 'ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links''IEEE/ACM International Conference on Automated Software Engineering'.

[96] Ncube, C.; Maiden, N.A.M.; Guiding parallel requirements acquisition and COTS software selection, Proc. IEEE International Symposium on Requirements Engineering 7-11 June 1999 Page(s):133 - 140

[97] Neumuller, C. & Grünbacher, P. (2006), 'Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned''IEEE/ACM International Conference on Automated Software Engineering'.

[98] Ngo-The, A. & Ruhe, G. (2003), 'Requirements Negotiation under Incompleteness and Uncertainty''Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering, San FranciscoBay (SEKE ’03), July, pp. 586-593'.

[99] Ni, D. C. (1997), 'Enumeration and traceability tools for UNIX and WINDOWS environments', Journal of Systems and Software 39.

[100] Ochs, M.; Pfahl, D.; Chrobok-Diening, G.; Nothhelfer-Kolb, B, A method for efficient measurement-based COTS assessment and selection method description and evaluation results, Proc. 7th Int. Software Metrics Symposium, 2001. METRICS 2001., 4-6 April 2001, Page(s):285 – 296

[101] Paulish, D. (2007), 'Methods, Processes & Tools forGlobal Software Development''ICGSE Remidi Workshop'.

[102] Paulk, M.; Curtis, B.; Chrissis, M. & Weber, C. (1993), 'Capability Maturity Model for Software', Technical report, CMU-SEI-93-TR-024, February.

[103] Pedreira, O.; Piattini, M.; Luaces, M. & Brisaboa, N. (2007), 'A systematic review of software process tailoring', ACM SIGSOFT Software Engineering Notes.

[104] Pierce, R. (1978), 'A Requirements Tracing Tool''Proceedings of the software quality assurance workshop on Functional and performance issues'.

[105] Pinheiro, F. (1996), 'Desighn of a Hyper Environment for Tracing Object-Oriented Requirements', PhD thesis, Wolfson College, University of Oxford.

[106] Pinheiro, F. & Goguen, J. (1996), 'An object-oriented tool for tracing requirements', IEEE Software 13.

[107] Pohl, K. (1996), 'PRO-ART Enabling Requirements Pre-Traceability''International Conference on Requirements Engineering'.

[108] Pohl, K.Ltd., J. W. R. S. P., ed. (1996), Process Centered Requirements Eng., John Wiley Research Studies Press Ltd.

[109] Radl, C., ‘A systematic literature review on requirements tracing approaches’, diploma thesis, Vienna University of Technology, 2007

[110] Ramesh, B. (2002), 'Process Knowledge Management with Traceability', IEEE Software 19.

[111] Ramesh, B. & Edwards, M. (1993), 'Issues in the development of a requirements traceability model''International Conference on Requirements Engineering'.

[112] Ramesh, B. & Jarke, M. (2001), 'Toward Reference Models of Requirements Traceability', Transactions on Software Engineering 58-9.

[113] Ramesh, B.; Powers, T.; Stubbs, C. & Edwards, M. (1995), 'Implementing Requirements Traceability; A Case Study''IEEE International Conference on Requirements Engineering'.

[114] Rational Unified Process, http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

[115] Richardson, J. & Green, J. (2004), 'Automating traceability for generated software artifacts''19th Int. IEEE Conf. on Automated SE, Linz, Austria, pp. 24-33'.

[116] Robertson S., The VOLERE process model, Internal Document, Atlantic Systems Guild, London, 1996,

[117] Romanski, G. (2003), 'Requirements, Configuration Management and Traceability for Safety Critical Software''IEEE International Conference on Requirements Engineering'.

[118] Ruhe, G. & Greer, D. (2003), 'Quantitative Studies in Software Release Planning under Risk and Resource Constraints''International Symposium on Empirical Software Engineering (ISESE ‘03), pp. 262-271'.

[119] C. Schwaber, “The Changing Face of Application Life-Cycle Management”, Forrester Research, August, 2006

[120] C. Schwaber, “The Expanding Purview of Software Configuration Management”, Forrester Research, July, 2005

[121] Siemens IT Solutions and Services, Requirements Engineer Seminar, Slides 2007

[122] Kaindl H., Lutz B., Tippold P., 'Methodik der Softwareentwicklung', Vieweg und Teubner, 1998

[123] Sommerville, I; Requirements Engineering – A good practice guide; Wiley, 1997

[124] Song, X.; Hasling, B.; Mangla, G. & Sherman, B. (1998), 'Lessons learned from building a web-based requirements tracing system"International Conference on Requirements Engineering'.

[125] Spanoudakis, G.; Zisman, A.; Pérez-Miñana, E. & Krause, P. (2004), 'Rule-based generation of requirements traceability relations', Journal of Systems and Software 72.

[126] Spence, I. & Probasco, L. (2000), 'Traceability Strategies for Managing Requirements with Use Cases', Traceability Strategies for Managing Requirements with Use Cases.

[127] Stone, A. & Sawyer, P. (2006), 'Exposing Tacit Knowledge via Pre-Requirements Tracing"IEEE International Conference on Requirements Engineering'.

[128] The Standish Group, Chaos Report, www.standishgroup.com,1995

[129] Thorp, J. et al.; The Information Paradox, McGraw-Hill, 1998

[130] Totz, G., 'Plug-in-basiertes Requirements Tracing in der Software Entwicklung', Diploma Thesis, Vienna University of Technology, 2007

[131] Travassos, G. H.; Mian, P. G.; Natali, A. C. & Biolchini, J. (), 'Experience on teaching and undertaking Systematic Reviews with graduate students at COPPE/UFRJ, COPPE / UFRJ', Technical report, Systems Engineering and Computer Science Program, Rio de Janeiro.

[132] Wahyudin, D.; Heindl, M.; Berger, R.; Schatten, A. & Biffl, S. (2007), 'In-Time Project Status Notification for All Team Members in Global Software Development as Part of Their Work Environments"International Conference on Global Software Engineering - SOFTPIT Workshop'.

[133] Watkins, R. & Neal, M. (1994), 'Why and how of requirements tracing', IEEE Software.

[134] Werner, C. M. (2003), 'OdysseyShare: an Environment for Collaborative Component-Based Development"IEEE Conference on Information Reuse and Integration'.

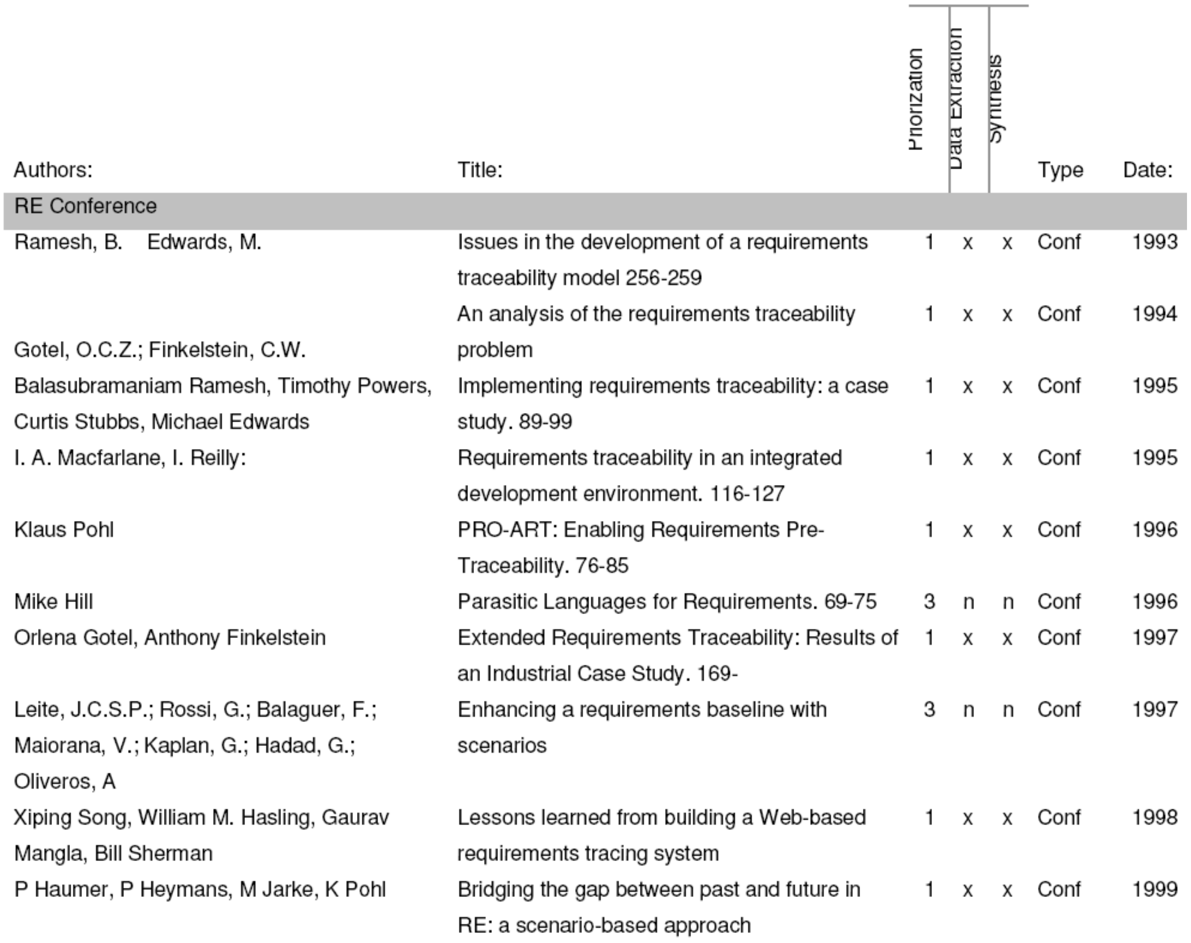
[135] Woodall, P. & Brereton, P. (2006), 'Conducting a Systematic Literature Review from the Perspective of a Ph.D. Researcher" 10th International Conference on Evaluation and Assessment in Software Engineering'.

[136] Yeoh, H.C.; Miller, J.; COTS acquisition process: incorporating business factors in COTS vendor evaluation taxonomy, Software Metrics, 2004. Proceedings. 10th International Symposium on 14-16 Sept. 2004 Page(s):84 - 95

[137] Yu, Y.; Wang, Y.; Mylopoulos and Liaskos, S.; Lapouchnian, A. P. & Leite, J. (2005), 'Reverse engineering goal models from legacy code"IEEE International Conference on Requirements Engineering'.

9 APPENDIX

9.1 Systematic Literature Review



Authors:	Title:	Priorization	Data Extraction	Synthesis	Type	Date:
RE Conference						
Ramesh, B. Edwards, M.	Issues in the development of a requirements traceability model 256-259	1	x	x	Conf	1993
Gotel, O.C.Z.; Finkelstein, C.W.	An analysis of the requirements traceability problem	1	x	x	Conf	1994
Balasubramaniam Ramesh, Timothy Powers, Curtis Stubbs, Michael Edwards	Implementing requirements traceability: a case study. 89-99	1	x	x	Conf	1995
I. A. Macfarlane, I. Reilly:	Requirements traceability in an integrated development environment. 116-127	1	x	x	Conf	1995
Klaus Pohl	PRO-ART: Enabling Requirements Pre-Traceability. 76-85	1	x	x	Conf	1996
Mike Hill	Parasitic Languages for Requirements. 69-75	3	n	n	Conf	1996
Orlena Gotel, Anthony Finkelstein	Extended Requirements Traceability: Results of an Industrial Case Study. 169-	1	x	x	Conf	1997
Leite, J.C.S.P.; Rossi, G.; Balaguer, F.; Maiorana, V.; Kaplan, G.; Hadad, G.; Oliveros, A	Enhancing a requirements baseline with scenarios	3	n	n	Conf	1997
Xiping Song, William M. Hasling, Gaurav Mangla, Bill Sherman	Lessons learned from building a Web-based requirements tracing system	1	x	x	Conf	1998
P Haumer, P Heymans, M Jarke, K Pohl	Bridging the gap between past and future in RE: a scenario-based approach	1	x	x	Conf	1999

Figure 47 Screenshot for the Excel Spreadsheet for the systematic literature review

9.2 Data Extraction Form

Research Topic

Short description: A Systematic Survey on Requirement Tracing Approaches

Estimate usefulness of the paper according to the research topic (0,1)

Review performed by Christoph Radl

Paper References:

Title:

author(s):

Keywords:

Conference Paper ☐ Journal ☐ ☐

Reference[1]:

Paper Classification:

	Yes	No
Experience Report:		
Case Study:		
Empirical Study:		

	Yes	No
Basic Paper:		
Concept Paper:		

Short summary of the paper

Quotes for Research Questions

RQ: TAF activities approaches implicitly/explicitly mentioned

Specification:

Generation:

Deterioration:

[1] Define year of publication and reference title (e.g., proceeding, book, journal)

Validation:

Rework:

Usage:

RQ: Parameters

RQ: Relationships between Parameters

RQ: Trace Activity Sets

RQ: Effects of RT approaches Precision Recall etc.

RQ: Open Issues

Comments:

Important Key references (selection):

9.3 Requirements Tracing Parameters

Dependency between parameters

For my traceability parameters I choose to partition them by the concept of an adaptive controller. The idea is that I have input and output parameters in the Tracing Activity Model. Within the TAF I have parameters that I can influence (tailoring parameters) and parameters I can just observe (working parameters). By this I can also support iterative options of the TAF and show how the change of tailoring parameters influences the working- and output parameters.

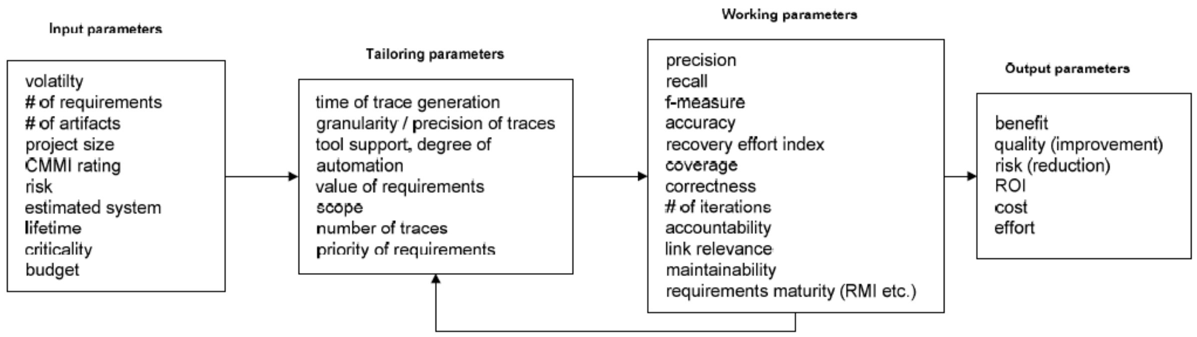


Figure 48 Dependencies between Requirements Tracing Parameters

Another interesting figure by Cleland-Huang et al. [3] also tries to show the different influences or tracing parameters.

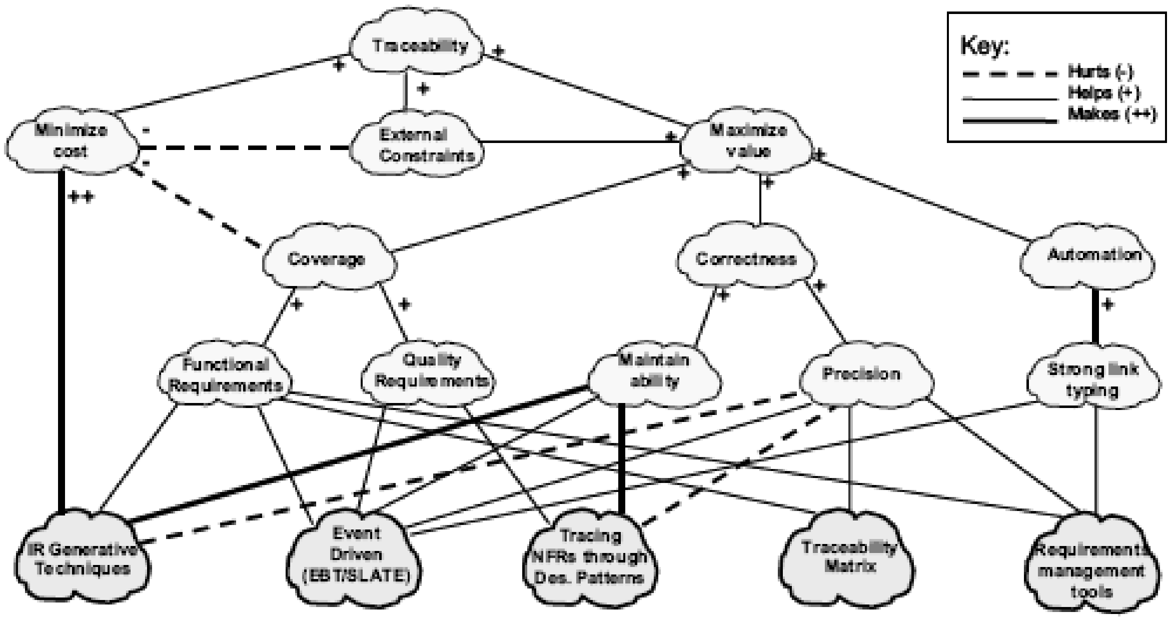


Figure 49 Softgoal Interdependency Goal showing traceability trade-offs [3]

Obviously this figure is not focussed on the parameters but on general goals for traceability. But these goals are often connected to the parameters I am going to describe Next I will introduce the parameters in detail. For each term I give the most common definitions I found in my retrieved literature. Some of the metrics are even ambiguous; in this case we give multiple definitions and try to agree on one. I also try to clarify the metrics with tangible examples to ease the understanding. Certainly it is also interesting to see how the different parameters influence each other. Thus I try to show dependencies at the explanations of parameters by referencing the dependent counterparts.

Input parameters (command variables)

Input parameters stand at the beginning of the tracing process. They are the basis for the decision whether and how to implement requirements traceability. Although they are usually not really influenceable we could think of cases where some of them are manipulated.

For example the CMMI rating of an organisation has to be taken as-is. But the project manager could postulate that process maturity must be brought to level two at least before starting to implement requirements tracing.

Another example could be the transformation of input parameter in output parameters: Through the introduction of requirements traceability "risk" is diminished as the integration phase is facilitated.

Table 28 Input Parameters for Requirements Tracing

Parameter	**Definition**
number of requirements / project size	Number of requirements is a primary measure in requirements tracing [83]. It determines which ways for requirements tracing are feasible and which ones are not. So for a small number they can easily be managed and traced with matrices documents and COTS software, but for a higher number tool support gets more important. The number of requirements is also a very important factor when recoding requirements inter-dependency. In the worst case we are facing an n2 complexity of requirements tracing (potential traces between all n artifacts) [63]. This poses performance questions about the needed tracing approach. Huffman-Hayes et al. [57] give the alternative term "size of domain". Currently 200-300 requirement object seem to be the maximum that a small team (5 persons) can handle comfortably [9]. So the number of requirements remains one of the most important input parameters for a requirements tracing approach.
Number of artefacts	An artefact can be a paragraph in the requirements document, method in a class, a test case etc. So it is interesting for us to see how many different artefacts we are facing in our project. Diversity of artefacts increases the effort for tracing.[59]
CMM(I) Rating / Process maturity	The Capability Maturity Model measures the quality (maturity) of software (development) processes. It has been replaced by the Capability Maturity Model Integration (CMMI) to prevent further uncontrolled growth of maturity models. In the case study of Ramesh et al. [67] it is mentioned that the management viewed requirements traceability as an important component in increasing their SEI CMM rating. It is necessary to implement requirements traceability to reach CMM level 3.
risk	Risk is often measured by the expectancy value for a potential

	negative impact on the project. So we have two values influencing the amount of risk. The probability value p and the magnitude of damage M. This definition can also be found in [23]. Example: If there is a 50% chance that we encounter an error that costs us 20000 EUR in our project the risk to happen is p*M therefore 0.5 * 20000 = 10000. In connection with requirements we can also look at the specific risk of a requirement. Heindl et al [63] mention the parameter risk of a requirement. They also suggest a three point scale with low medium and high risk as values. Generally the risk is often dependent on the à volatility of a requirement.
Required safety /criticality	Another important measure is the degree of security relevance of the system. Here a scale high, medium and low is possible. A system with high security relevance might be software to control medical devices or the core system of a bank. Low security relevance might be found in the domain of entertainment software etc. When encountering a system which is highly safety-critical requirements traceability is very recommendable [133]. Another term for this could be "criticality" which is mentioned by Cleland Huang et al. in [56].
estimated system lifetime	influences the decision whether or how to trace requirements. Kotonya et al. [83] state that more comprehensive traceability policies should be defined for systems with a long lifetime. Generally the real system lifetime is often longer than the time estimated
Complexity	Means the complexity of trace objects [63]. But we could also look at the complexity of the system as a whole. A popular approach is to express the complexity of project in function points. In the work of Ramesh et al. [112] the perceived importance of traceability grows with the system complexity
volatility / probability of change	According to Kotonya et al. [83] volatility is a measure of how much requirements change over time. Some kind of volatility of requirements is natural and not necessarily due to poor requirements elicitation. However it is obvious that it is desirable to keep volatility low. We have already introduced factors influencing the volatility in section 1.6.1. A synonym for volatility is the probability of change (PC) value introduced in [23]. They use a scale form 0-1, where 9.0 is "volatile" 0.6 is "changeable" and 0.3 is "stable".
Failure to trace	Additionally to volatility / probability of change we find the measure Failure to Trace (FTR) in [23]. The probability of change is here enhanced with the impact a change in one requirement has on the whole system. This indicates the danger of NOT tracing the respective

	requirement. The idea is that volatility itself is not a problem if the changing requirement is rather unimportant. The formula for FTR is: FTR = ((PC × 2)+1)× I Where PC means the probability of change and I is Impact.
Budget	Budget is a main influence whether requirements traceability can be introduced or not. Especially if it is newly introduced in an organisation budget can be a very sensitive issue

Parameters of the tracing process

So the input parameters influence the decision about the tracing approach. When we have chosen such an approach, we are facing additional parameters – those of the approach itself. They result from the decision how to trace. Some of them can be adjusted during the tracing process while others have to be fixed at the beginning of the process.

Tailoring parameters

The tailoring parameters have to be set mainly in the specification phase. Once they are set, they cannot be changed easily or just by a second iteration of all tracing activities. Actuating variables influence the later parameters and the whole set of activities.

Table 29 Tailoring Parameters for Requirements Tracing

Parameter	**Definition**
time of trace geneneration, delay between artefact- and trace generation	It specifies when trace information is being generated [7]. Here the values of the metric could be connected to the stages of the software engineering process. Also from [7] we get the hypothesis that capturing traceability information in earlier phases of the software development lifecycle is much easier than in later phases. Example: We can think of generating trace information manually during the programming of a system. But it is also possible to generate trace information later after the development by an after-the-fact tracing approach. With manual trace generation we are facing an earlier point in time of trace generation than with the automated approach. Another way is to generate traces manually after programming this tends to be a very time consuming and cumbersome duty. A desirable point is to generate trace information transparently in real time during programming. However this requires integrating a trace generation tool directly in the development environment. So here we do not have discrete values, but relative ones. Or we are facing development phases as values. Possible values could be • Continuous (real-time) • After the fact
Granularity / precision of traces	Granularity describes how detailed to trace requirements [7]. So it tells us whether we are tracing at package, class or method level. In Heindl et al. [63] we also find concept of a measure for granularity, but they refer to it with the term “precision” meaning how precise to trace the requirements/artifacts. Granularity is connected with two factors. It depends on the way of implementation. E.g. if all requirements are implemented in very few classes

	within one package we need high granularity anyway to trace to any of the requirements. From a value based perspective it is beneficial to trace more important, volatile, risky requirements with a higher granularity than others because the trace information has more value as we might need it more often. [63] Furthermore empirical studies of Egyed et al. show that tracing on method level is about 10 times more expensive than on class level [36]. It is important to say that an increase in granularity offer a more detailed picture on the system on one hand, but also leads to an increase of -> Cost. This is a trade off one has to made when specifying the level of granularity.
Tool support	Mentioned in Heindl et al. [63]. Tool support is a metric that is often hardly to measure with numbers. One could just formulate, that for one kind of requirements implemented in a specific technology, there are tools available to trace them automatically. Example: if you can trace into java code and 30% of your requirements are implemented in java the tool support would be equal to this.
Degree of automation	Is defined as the extent to which a traceability link supports automated queries. [23] Potential values are • non-automated, • semi-automated • fully-automated Non-automated links require the user to manually traverse the links to identify related artefacts. Semi automated links support queries that return lists of artefacts meeting the criterion of the query, while fully-automated links provide much higher-level support for utilizing the links to identify impacted artefacts, and for transmitting data and commands that trigger events within the impacted artefacts.
Number of artefacts to be traced	Refers to the different entities between which we are generating traces. If the number of artefacts to trace gets higher the effort also rises to create traces between them [63]. This term is connected to the “number of requirements”. So we could look at the number of artefacts to be traced in general. Or we could ask how many artefacts per requirement have to be traced.
Value of requirements	In [63] we find the term value of requirements. In this work the value is simplified to a three point scale. The value represents the value to the customer; here exist the categories 1) low value, 2) medium value, 3) high value
Value of traces	With this measure we can quantify the value of single traces [63]. It is clear that it is dependent to the value of requirements
Scope	refers to just taking a subset of the artefacts for requirements tracing [63]. For example we could formulate to reduce the scope of tracing to the business logic of our system, leaving other parts untraced
Number of traces	refers to the number of relations we create from, to or between requirements [63]. It is connected to the measure à number of artefacts
Priority of requirements	should be included in the data scheme for each requirement. It defines the importance of a requirement. Various metrics are possible: from high to low or a number scale from 1 to 10

Working parameters
Working parameters can be observed from the trace generation on. They are influenced by the configuration of the tailoring parameters. Their observation allows us to rate how well our tracing approach works. I employed subcategories for the working parameters to keep an overview. First I have the general working parameters, then the information retrieval parameters and finally the requirements maturity parameters.

Table 30 General Working Parameters for Requirements Tracing

Parameter	Definition
Number of queries	Describes how often a specific trace has been queried [8]. Example: Out of two traceability links A and B, the link A is queried two times. B in comparison is queried 30 times. This gives us important information about the importance of links or can even be used to predict potential problems. So this measure is some kind of logging information to see how often a specific trace is requested. We can summarise that the number of queries to trace a specific requirement can be used for "hot spot" location.
Correctness of traces	Correctness is the "extent to which a set of links conforms to the actual relationships that exist between artefacts" [23]. So it is a measure how well the real world has been represented in the traceability model. Example: If we have 200 forward-to traces, but currently our tracing infrastructure maintains only 180 of them, the correctness value for our approach would be 90%. Within the manual trace generation approaches, correctness is also referred to as a synonym for precision [59]. I.e. correctness is 100% if there are no more false positive traces left.
Coverage	Coverage is "the extent to which the traceability scheme provides support for all requirements regardless of their type, geographical location, or level of abstraction" [23]. Romanski [117] in comparison states that "coverage in this case means that all of the code and all conditions were exercised" Therefore we can summarize that coverage is a measure how many different types of artefacts can be traced. Of the percentage of coverage depends on how many artefacts are available. So if there is only source code residing on one machine to trace and the approach traces it the coverage of this approach would be 100% Example: The tracing infrastructure allows tracing to source code but no tracing to the author of a requirement. Now from 50 requirements you trace to 50 points in source code (forward to). But starting from the requirements it would be important to trace back to the authors of the 50 requirements (backward from). So the coverage is just 50% in this case. Another example could be that it is impossible to trace to requirements implemented in a branch office. Thus if the branch office is responsible for 20% of the implementation work, coverage drops to 80%.
Number of iterations	Iterations are used as a measure, especially for methods, algorithms which work incrementally, or to see the evolution of an algorithm [70]. We see that there can be multiple iterations of the activities between generation and usage (including them). Trace specification might also be part of iteration but usually the process is defined once. From Hayes et al. we learn that a higher number of

	iterations generally lead to higher recall and precision [56].
average effort per unit to be traced	Starting from the known total cost we divide it by the number of units, requirements or simply artefacts and get an arithmetic mean value. Example: From historical data we know that requirements tracing costs us additional 10% of workload. So at a 10 man-month project requirements tracing costs us an additional man-month. Saying that this month includes 160 hours we have a more detailed view. By multiplying it with the cost of work and adding potential fixed costs for the introduction of a traceability infrastructure we get the total tracing effort say 20000 EUR. If we have 200 artefacts our average effort per unit would be 100 EUR. It is important to say that the average effort does not properly scale in many cases because for a higher number of units we need other approaches. So the average effort should only be used in comparable settings or to show the differences between several approaches.
number of contributions per author	Gotel and Finkelstein introduce special responsibility metrics in [50] e.g. number or contributions per author. So these metrics are personbased. The person is somehow also seen as an artefact to which and from which one can trace requirements. Obviously we are facing discrete values in this case.
Accountability	In [133] Watkins an McNeal assume that "the project will have a better success rate if the data is available for auditors and you can prove that a requirement was successfully validated by associated test cases." So accountability tells us to which degree we are able to prove that our system fulfils all the requested duties.
Strength of trace link candidate	Egyed et al. [36] introduces the measure strength of trace link candidate. It is defined as the ratio of the number of code elements that implement a requirement and the number of code elements that two code elements share as a part of their implementation. Example: A requirement X is implemented in 10 methods and another requirement Y is implemented in 5 methods. 2 methods implement both requirement X and requirement Y. Strength of X = 2/10 = 20% Strength of Y = 2/5 = 40% Stronger trace links are less likely to be false positives [36].The authors mandate to examine especially requirements with low strength as they state that requirements with zero strength and very high strength are very likely to be correct. We further learn that when employing thresholds on the basis of the strength to eliminate false positives, some true links might also be affected. Therefore these strength-thresholds can hurt the -> correctness of the traceability scheme
Maintainability	Is described as the ease by which an existing set of correct traceability links can be preserved and evolved in an accurate state [23] [133]. It indicates how much effort is necessary to maintain the traceability approach. Here also more general values like low, medium and high are possible.

Information retrieval parameters

After-the-fact tracing approaches need a special subset of parameters. A branch of research called information retrieval; IR [46] deals explicitly with this topic. The general aim is to quantify how well your technique to obtain results works. The information retrieval parameters are relevant for after-the-fact generation of traces we therefore pay attention to them during "trace generation".

Table 31 Information Retrieval Parameters

Parameter	**Definition**
Recall /completeness	Recall is a term primary term of information retrieval. It defines the completeness of a search conducted against a sample. It measures the number of correctly retrieved documents out of the entire set of correct documents [23]. Mathematically recall is Correct links found/ Correct links. The concrete meaning of recall in the context of automated trace generation is which percentage of the traceability links which should exist have been generated by the algorithm. Here we also want to give the formula: x = relevant documents found z = relevant documents not found recall =x/ (x+ z) So recall tells us the percentage how many of the relevant documents we have found. The problem with this measure is that the number of total relevant documents (which is necessary to calculate recall) is often unknown. Therefore we have to refer to the "relative recall". Example: Knowing that there are 80 traceability links, which are interesting for our specific problem we run our approach and get 72 links. In this case our recall value would be 0.9. This would mean that our approach has omitted 0.1 or 10% of the relevant links.
Precision	Precision is a term derived for information retrieval as well. It defines the rate of relevant results in a search. Thus for a search it tells us the percentage of relevant results. In the context of automated trace generation precision tells us how many useful links have been generated in the retrieved set. [21][56]. Another definition is correct links found/ Total Number of Candidates [57]. The precision value therefore indicates which percentage of the links we have to sort out after the generation because they were false-positives. Again we want to give the formula as well: x = relevant documents found y = irrelevant documents found precision = x/(x+ y) Example: Continuing out example from above, we have found 40 traceability links. During validation we discover that 30 of them were irrelevant for our query. Applying the formula above our precision is 0.25.
Accuracy	In [56] we find accuracy as a quality measure for the retrieval of traceability links. It is determined by recall and precision while these two values are often connected in an orthogonal way. This means a high recall often leads to low precision. On the other hand a low recall means that many potential links are outside of the retrieved set of links although precision might not be bad in these cases, there a still links outside the retrieved set left which must be examined. There is no explicit formula for the calculation of accuracy (f-measure). So we

	summarize that accuracy is the extent to which a requirements tracing tool returns all correct links and the extent to which the tool does not return incorrect links.
f-measure	is a harmonic mean of precision and recall. It measures the overall performance of an information retrieval approach. The formula is: f-measure = 2*(precision*recall)/(precision+recall) Example: In the examples above we calculated a recall value of 0.5 and a precision value of 0.9 so inserted in the formula above we get an f-measure of 0.6428. Of course we want to have a high f-measure with our approach but achieving high precision and high recall is a balancing act (as precision increases, recall tends to decrease and vice versa.) [70] Therefore f-measure is a useful indicator for assessing and comparing the quality of tracing approaches.
Link Relevance	While calculating precision and recall we only spoke of relevant and irrelevant links. But the relevance can also be expressed by a percentage value. In [56] not every true link is always 100% relevant while a false link is not always completely irrelevant. As a synonym link probability is used by Lin et al. [89]. In their case this is the probability that the automatically retrieved link is relevant.
ART	Average relevance of a true link in the list; this is the arithmetic mean of all relevance coefficients of true links [56]. Example: Having three true links with the relevance coefficients of 1.0, 0.75 and 0.5 our ART would be 0.75
ARF	Average relevance of a false positive in the list; this is the arithmetic mean of all relevance coefficients of true links [56]. Example: Having three true links with the relevance coefficients of 0.0, 0.05 and 0.1 our ART would be 0.05
DiffAR	DiffAR measures the the difference between average relevances of true links and false positives [56]. So the formula is DiffAR = ART –ARF. Example: Remembering the ART of 0.75 and the ARF of 0.05 from above our DiffAR would be 0.7
Lag	Lag [56][70]is the average number of false positives with higher relevance coefficient than a true link. (the number of false-positives that are higher up in the list of retrieved requirements) Example: Continuing the example from ART and ARF above we remember our true links 1.0 0.75 and 0.5 and our false links 0.0 0.05 and 0.1. If we introduce two false links with a relevance of 0.8 and 0.81 (which are both higher than the lowest true link) our lag value is 2. It is obvious that we aim to minimize the lag. For this the information retrieval algorithm has to be improved over time.
Selectivity	In general, when performing a requirements tracing task manually, an analyst has to examine a high number of candidate links, i.e., perform an exhaustive search. Selectivity measures the improvement of an IR algorithm over this number. The lower the value of selectivity, the fewer links that a human analyst needs to examine [70].
Recovery Effort Index (REI)	According to Antoniol et al. [4] REI is defined as the ratio between the number of documents retrieved and the total number of documents available. The lower the REI, the higher the benefits of the IR approach. Is actually a synonym for for recall.

	REI = (#Retrieved/#Available)%

Maturity parameters
The requirements maturity index [9][45] is a family of parameters which aim to quantify the change occurring to the requirements over time. With different ways it measures the à volatility of a project as a whole. I could also measure the volatility for just a subset of requirements which is of speacial interest. Of course the changing of requirements affects the traces as well, therefore are the maturity parameters relevant for tracing. A perquisite for calculating maturity parameters is that multiple iterations of the tracing activity model are carried out.

Table 32 Requirements Maturity Parameters

Parameter	**Definition**
RMI	The Requirements Maturity Index (RMI) is described by Felici [45] in detail. He defines it as an indirect measure that relies on two primitives (or direct measures): RT and RC. RT is the total number of software requirements in the current release. RC is the number of requirements changes, i.e., added, deleted or modified requirements, allocated to the current release. RMI = (RT-RC)/RT Obviously the RMI is only useful if it is calculated for each software release, so trends concerning the maturity of the system can be derived. Also Arkley et al. found the RMI a very useful measure for their project in [9]. There is some following key data connected to the RMI, like the Historical Requirements Index (HRMI), the Ageing Requirements Maturity (ARM) or the RSI, which we introduce below. Example: In the current release are 150 requirements. 20 of them changed. Applying our formula above the RMI is 0.86.
Cumulative number of requirements changes	Or CRc [45] measures the number of total changes of requirements over all releases.
Average number of requirements change	Or ARc [45] is the CRc divided by the number of iterations n. So for 100 changes over 5 iterations our ARc would be 20.
Requirements stability index	Or RSI [45] measures how stable the requirements are over time. Remembering Rt the total number of requirements and CRc the cumulative number of requirements changes the formula is: RSI = (RT-CRc)/RT

Output parameters
Requirements tracing is done to receive results. These results are quantified by the output parameters. Some of the input parameters can reappear as output parameters because they have been manipulated by the requirements tracing process.

Table 33 Output Parameters

Parameter	**Definition**
Benefit	Is the effort saved through requirements tracing [8] [9][67] [21]. Though it not always possible to give exact quantitative examples for the benefit of requirements tracing we try to give an example: If the normal maintenance effort during the project (e.g. searching for classes where a specific feature has been implemented) is 10% of the total development cost and it can be reduced to 8% by requirements tracing the benefit is 2% of the total development cost.
Quality	Quality can be described as the achievement or excellence of an object. In software development quality can be described as the absence of errors. So a higher quality, or reduced maintenance effort (long term) is the result.
Risk Reduction	Arkley et al. [9] refer to risk and its reduction as a measure for the success of requirements tracing. It is defined as the ability to identify development components which could be reused resulted in a perceived reduction in project risk.
ROI	Descibes the amount of money gained relative to the amount of money invested. This “gain” can also be a saving of cost, but in this case there is no explicit out payment. The ROI is measured as percentage. Example: Starting from an investment of 1000 currency units the investments offers 20 additional currency units. So the ROI is 2% in this case. In finance the ROI must often compete with an alternative guaranteed financial investment [59].
Effort	Is obviously the amount of work that has to be done either to enable or to enact requirements tracing. There are multiple ways to measure effort. It is very common to measure effort in time. Examples are man-year or man month. For smaller scales hourly measures are also possible. Of course there is not just time but money involved as well. For this I introduce the measure “cost” below. Example: if a project takes ten man-months without tracing, but we know that the introduction of requirements traceability generated 10% additional workload our effort for tracing would be 1 manmonth.
Cost	Is also often found in literature, it is the financial representation of effort. As consumed time leads to cost we can multiply the timemeasure (such as month or hour) with the costs for staff and infrastructure per unit to calculate the cost. But cost is not just the cost for the maintenance of operation. Cost can also be a loss of revenue. As the product does not generate revenue in case of a delay in the release the revenue not received is cost as well [18].

10 Curriculum Vitae Matthias Heindl

PERSÖNLICHE DATEN

Name	Matthias Heindl
Geburtsdatum	21.01.1980
Geburtsort	Wien
Staatsbürgerschaft	Österreich
Familienstand	Ledig
Adresse	Maurer Lange Gasse 39-41/1/6, 1230 Wien
Telefon	0699 1236 1644
E-mail	matthiasheindl@gmx.at

AUSBILDUNG

Mai 2008	Zertifizierung zum „Certified Professional for Requirements Engineering“
Dez. 2004	Beginn des Magisterstudiums der Wirtschaftsinformatik, TU Wien
Nov. 2004	Beginn des Doktorat-Studiums Informatik, TU Wien
Nov. 2004	Abschluss des Studiums der Informatik, TU Wien Diplomarbeit: „Evaluating a Value-based Requirements Traceability Approach in a Real-life Project Setting“ in englischer Sprache
März 2003	Inskription des Wirtschaftsinformatik-Studiums, TU Wien
Jan. 2002	Abschluss des 1. Studienabschnitts der Informatik, TU Wien
März 1999	Inskription des Informatik-Studiums, TU Wien
Juli 1998 – Feb.1999	vollständige Absolvierung des Präsenzdienstes beim österreichischen Bundesheer, Jägerregiment 2, Maria Theresia Kaserne
Sept. 1990 – Juni 1998	Realgymnasium im Kollegium Kalksburg, 1230 Wien, mit abschließender Matura im Juni 1998

BERUFSERFAHRUNG

November-Februar 2008	SIEMENS Einsatzbereich: Siemens Transportation Systems Aufgaben: Requirements Management Consultant für das ÖBB Projekt Railjet (Teilprojekt Führerstand Leittechnik-Software)
Seit September 2007	TU WIEN Technische Universität Wien Vienna University of Technology Einsatzbereich: Institut für Softwaretechnik der TU Wien Aufgaben: Lehrbeauftragter für die Lehrveranstaltung „Requirements Analyse und Spezifikation“

Mai-September 2007

SIEMENS

Einsatzbereich: Projektleiter des technischen Teilprojekts zur Verbesserung des SIEMENS PSE-internen Software-Entwicklungsprozesses

Aufgaben:

Projektleitung, Verwendung des Eclipse Process Frameworks (EPF) zur Modellierung des Softwareentwicklungsprozesses

Seit März 2006

Einsatzbereich: Institut für Softwaretechnik der TU Wien

Aufgaben: Lehrbeauftragter für die Lehrveranstaltung „Risikomanagement"

Seit Juli 2005

SIEMENS

Einsatzbereich: Siemens Programm- und Systementwicklung (PSE) KB C5, Consultant im Support Center Configuration Management

Aufgaben:

Mitenwicklung eines Requirements Engineering Kurses,

Requirements Engineering Training,

Configuration Management (CM) Training

CM Tool Trainings (Subversion)

Requirements Management Prozess-Consulting und Support für Projekte

Requirements Management Tool Trainings (RequisitePro, Doors)

März 2003 bis März 2006

Einsatzbereich: Institut für Softwaretechnik der TU Wien

Aufgaben: Tutor für die Lehrveranstaltung „Risikomanagement VU", Planung und Durchführung der Lehrveranstaltung „Risikomanagement VU": Abhaltung von Workshops zur Risikoidentifikation, Risikoanalyse und Gegenmassnahmen-Entwicklung

Aug. und Sept. 2003

SIEMENS

Einsatzbereich: Siemens Programm- und Systementwicklung (PSE) AS TT

Aufgaben: Datenmigration der Testdaten für Make.IT Demonstrator und Test dieser Daten, Review des Pflichtenheftes von Make.IT mit Augenmerk auf Requirements Tracing, Entwurf und Umsetzung eines Bewertungsverfahrens zur Kategorisierung von Requirements, Diplomarbeitsbeginn (Vorarbeiten: Recherche)

Okt. 2002 bis März 2003

Einsatzbereich: Institut für Softwaretechnik der TU Wien

Aufgaben: Tutor für die Lehrveranstaltung „Qualitätssicherung VU", Erstellung und Korrektur der Übungsaufgaben (Blackbox- und Whitboxtests), Erstellung einer eigenen Lehrveranstaltungs-Website, Planung der neuen Lehrveranstaltung „Risikomanagement VU", Folienerstellung, Skriptumerstellung

Aug. und Sept. 2002

SIEMENS

Einsatzbereich: Siemens Programm- und Systementwicklung (PSE) AS TT

Aufgaben: Implementierung einer beispielhaft vorgegebenen Anbindung an ein externes System im Rahmen der Akquisition und bei der Adaptierung dieser Schnittstelle zur Test-Verbesserung. Arbeit mit Java, JSP, Perl, XML, XSL,

	HTML und Oracle
Sept. 2001	SIEMENS Einsatzbereich: Siemens Programm- und Systementwicklung (PSE) AS TT Aufgaben: Programmierung eines Webapplikations-Prototypen in Java im Rahmen des Projekts „MAKE IT“
Aug. und Sept. 2000	Dräger DRÄGER Austria, Wien Einsatzbereich: Abteilung Export Aufgaben: Businesspläne-Layout überarbeiten, Artikel in SAP anlegen
Februar 2000	Dräger DRÄGER Austria, Wien Einsatzbereich: Abteilung Export Aufgaben: Artikel in SAP anlegen, Gestaltung der Businesspläne (Excel)
Juli 1999	Dräger DRÄGER Austria, Wien Einsatzbereich: Abteilung Medizingeräteverkauf Aufgaben: Überarbeitung von Präsentationsdateien, Datenbankerstellung für Referenzlisten mit MS Access, Überarbeitung der Kundendatenbank

FREMDSPRACHEN

Englisch	Fließend in Wort und Schrift
Französisch	Anfänger/Grundkenntnisse
Tschechisch	Anfänger/Grundkenntnisse

FACHLICHE KENNTNISSE

Programmierung	Java, SQL, XML, XSL, UML Erfahrungen mit Perl, C
Web Engineering	JSP, PHP, Flash, Apache Web Server, Tomcat
Relationale DBMS	Oracle, MySQL, PostgreSQL
Betriebssysteme	MS Windows, Grundkenntnisse in Linux/Unix
Anwendungen/Tools	MS Office, MS Project, MS Visio, Rational Rose, Rational XDE, Rational Requisite Pro, Telelogic Doors, Subversion, Dreamweaver, Latex, Eclipse Process Framework (EPF C)
Projekt- und Qualitätsmanagement	Softwareprojekt-Risikomanagement, Requirements-Analyse, Requirements Tracing, Requirements Negotiation, Configuration Management, Capability Maturity Model (CMM), Methoden der Qualitätssicherung (Tests und Reviews)

PUBLIKATIONEN

2007	Alex Egyed, Paul Grünbacher, Matthias Heindl, Stefan Biffl, Value-Based Requirements Traceability: Lessons Learned, RE'07: 15th IEEE International Requirements Engineering Conference, India Habitat Center, New Delhi, October 15-19th, 2007
	Dindin Wahyudin, Matthias Heindl, Benedikt Eckhard, Alexander Schatten and Stefan Biffl, In-time role-specific notification as formal means to balance agile practices in global software development settings, 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET) 2007, Poznan, Poland
	Matthias Heindl and Stefan Biffl, A Framework to Balance Tracing Agility and Formalism; 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET) 2007, Poznan, Poland
	Matthias Heindl, Franz Reinisch, and Stefan Biffl, Requirements Management Infrastructures in Global Software Development - Towards Application Lifecycle Management with Role-based In-time Notification, International Conference on Global Software Engineering (ICGSE), Workshop on Tool-Supported Requirements Management in Distributed Projects (REMIDI), August 2007, Munich
	DinDin Wahyudin, Matthias Heindl, Ronald Berger, Stefan Biffl, Alexander Schatten, In-Time Project Status Notification for All Team Members in Global Software Development as Part of Their work environments, International Conference on Global Software Engineering (ICGSE), Workshop on Measurement-based Cockpits for Distributed Software and Systems Engineering Projects (SOFTPIT), Munich, August 2007
	Armin Beer, Matthias Heindl, Issues in Testing Dependable Event-Based Systems at a Systems Integration Company, 2nd International Conference on Availability, Reliability and Security (ARES), Vienna, 2007
2006	Matthias Heindl and Stefan Biffl, The Impact of Trace Correctness Assumptions, 5th ACM/IEEE International Symposium on Empirical Software Engineering 2006, Rio de Janeiro, September 2006
	Heindl, M.; Reinisch, F.; Biffl, S.; Egyed, A; Value-Based Selection of Requirements Engineering Tool Support; 2006. 32nd EUROMICRO Conference on Software Engineering and Advanced Applications 29-01 Aug. 2006 Page(s):266 - 273
	Matthias Heindl, Stefan Biffl; Project management: Risk management with enhanced tracing of requirements rationale in highly distributed projects, May 2006 Proceedings of the 2006 international workshop on Global software development for the practitioner GSD '06
2005	Alexander Egyed, Stefan Biffl, Matthias Heindl, Paul Grünbacher, A value-based approach for understanding cost-benefit trade-offs during automated software traceability, November 2005 Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering TEFSE '05
	Alexander Egyed, Stefan Biffl, Matthias Heindl, Paul Grünbacher, Determining the cost-quality trade-off for automated software traceability, November 2005 Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering ASE '05
	Matthias Heindl, Stefan Biffl, A case study on value-based requirements tracing, September 2005 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering ESEC/FSE-13

S. Biffl, M. Heindl, M. Auer, and M. Halling, Tool Support for a Risk Management Process - An Empirical Study on Effectiveness and Efficiency, Software Engineering SE 2004, 2/17/2004 - 2/19/2004, Innsbruck, Austria

SONSTIGE AUSBILDUNGEN UND PRIVATE AKTIVITÄTEN:

Sport	Ausbildung zum staatlich geprüften Tennislehrer
	Ausbildung zum ÖTV-Coach – 1.Teil
	Seit Jahren Teilnahme a. d. Wiener Mannschafts-Meisterschaft für den TC Kalksburg in der Landesliga A
	Mannschaftsführung einer Tennis-Jugendmannschaft
	Nebenberufliche Tätigkeit als Tennislehrer (Organisation und Durchführung von Jugendtenniscamps, Jugend-Trainings, Trainings für Erwachsene etc.)
	Fussball im Wiener Fussballverband (SU Kollegium Kalksburg)
	Schifahren, Langlaufen, Joggen, Snooker, …
Sonstiges	Lesen, Jazz, Kino, Klavier, Philosophie