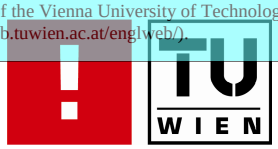

FAKULTÄT FÜR !NFORMATIK

Design and Development of a Traceability Framework for Small Software Development Teams – A Message-Oriented Approach

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Martin Schwarzbauer
Matrikelnummer 9725707

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Grechenig

Wien, 4.11.2008

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Design and Development of a Traceability Framework for Small Software Development Teams – A Message-Oriented Approach

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Martin Schwarzbauer

9725707

ausgeführt am

Institut für Rechnergestützte Automation

Forschungsgruppe Industrial Software

der Fakultät für Informatik der Technischen Universität Wien

Betreuung:

Betreuer: Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Grechenig

Mitwirkung: Mario Bernhart

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 4.11.2008

Martin Schwarzbauer

Danksagung

Ich möchte Professor Grechenig danken, vor allem dafür dass er mich in der Anfangsphase meiner Diplomarbeit ermutigt hat, an meiner Idee einer Traceability-Lösung für kleine Softwareentwicklungsteams zu arbeiten, sie zu detaillieren und zu erweitern.

Mein Dank gilt auch Mario Bernhart, der mir in den entscheidenden Phasen geholfen hat, meine Anstrengungen auf ein erreichbares Ziel zu konzentrieren und mich dadurch davor bewahrt hat, mich mit mehr Problemen zu beschäftigen, als ich im Rahmen meiner Arbeit lösen hätte können.

Ich bin Brigitte Brem sehr dankbar für Ihre Anmerkungen und Anleitung während der abschließenden Arbeiten an dieser Diplomarbeit, ohne die ich nicht imstande gewesen wäre, den letzten Einreichtermin für meinen Studienzweig zu halten.

Meiner Lebensgefährtin Yvonne danke ich für ihren Hinweis, dass es allein meine eigene Schuld war, dass mein Abschluss aufgrund des Termindrucks fraglich war, und vor allem dafür, dass sie mich mich trotz dieser Tatsache in Liebe unterstützt hat.

Zum Abschluss möchte ich auch meinen Eltern danken, die mir immer ein Rückhalt waren und mich auch dann unterstützt haben, als meine Studienfortschritte aufgrund mangelnder Konsequenz und Anstrengung meinerseits seltener wurden.

Acknowledgements

I want to thank professor Grechenig, especially for encouraging me in the inception phase of the thesis to work on, detail and expand my idea of a traceability solution for small development teams.

My thanks also go to Mario Bernhart, who helped me focus my efforts on an achievable goal at the decisive stages of my thesis and thus preserved me from trying to solve more problems than I could have handled in this thesis.

I am very grateful to Brigitte Brem for her feedback and guidance during the finalisation of my thesis, without which I would not have been able to hand in my work in time for the last chance to graduate in my branch of study.

To my significant other Yvonne, I owe thanks for kindly pointing out to me that getting into a situation where my graduation was in peril was entirely my own fault, and even more gratitude for showing unwavering love and support in spite of that fact.

Finally, I want to thank my parents, who have always backed me up and strongly supported me throughout my entire studies, even when there were few results due to little effort on my part.

Kurzfassung

In der Softwareentwicklung werden Verfolgbarkeit des Informationsflusses und Nachvollziehbarkeit von Entscheidungen (Traceability) oft eher als bürokratisches Hemmnis denn als Möglichkeit zur Effizienzsteigerung gesehen. Dadurch sind gerade kleinere Softwareentwicklungsteams kaum motiviert, die genannten Konzepte anzunehmen und einzuführen. Diese Arbeit hat das Ziel, eine akzeptable Lösung für das Verknüpfen von Projektzwischenergebnissen (Artefakten) zu präsentieren – eine Methode, um Applikationen in der Softwareentwicklung zu integrieren – um es auch kleinen Softwareentwicklungsprojekten zu ermöglichen, Traceability mit einem vertretbaren Aufwand zu erreichen und daraus Vorteile zu ziehen.

Das Hauptproblem ist die Tatsache, dass verschiedenste Applikationen zur Unterstützung der Softwareentwicklung im Einsatz sind, sodass ein gemeinsamer Kommunikationsstandard etabliert werden muss, um den Informationsaustausch zwischen beliebigen Zusammenstellungen dieser Applikationen zu ermöglichen. Der hier präsentierte Vorschlag ist ein XML-basiertes Nachrichtenformat: Nachrichten mit Informationen zur Identifikation von Artefakten sowie Nachrichten, die Beziehungen zwischen jeweils zwei Artefakten definieren.

Der Einsatz von Nachrichten anstelle von Schnittstellen erlaubt eine lose Kopplung der Applikationen, sodass diese integriert werden können, ohne die applikationsinternen Konzepte anzutasten. Dadurch werden schnelle Resultate erzielt, etwa durch Erweiterung einer Anwendung, die somit Nachrichten an eine zweite Anwendung verschickt, die diese verarbeitet und optisch für Benutzer aufbereitet. Dadurch ist der erste Schritt Richtung Traceability umgesetzt, unter verhältnismäßig geringem Aufwand. Die Verwendung von XML als Basis ermöglicht das automatische Validieren von Nachrichten sowie das Erweitern um zusätzliche Informationen, um eine engere Zusammenarbeit zwischen einzelnen Applikationen zu ermöglichen.

Als Machbarkeitsstudie wurden zwei Applikationsprototypen entwickelt – eine für Anforderungsmanagement und eine für Testverwaltung – und diese so erweitert, dass sie durch das präsentierte Nachrichtenformat kommunizieren. Das System zeigt, dass damit Verknüpfungen über Anwendungsgrenzen hinweg erstellt werden können und dass dadurch ein Mehrwert gegeben ist. Aufgrund der ausgetauschten Informationen könnten zum Beispiel Berichte über Testabdeckung oder Testergebnisse zu einzelnen Anforderungen erstellt werden.

Keywords: *Verfolgbarkeit, Softwareentwicklung, Tool-Integration, XML-Nachrichten, dezentrale Verknüpfungen*

Abstract

Traceability in software development is often seen as bureaucratic burden rather than an opportunity for efficiency improvements. Consequently, small development teams are hardly motivated to embrace the concept and make information flow and decision making in their projects traceable. This thesis aims to present a viable solution for linking intermediate products (artefacts) of the project – a method of integrating tools used in the development process – so that small development teams can reap the benefits of traceability with a reasonable amount of effort.

The primary issue is the fact that there are various tools available and in use – a common standard of communication is needed to enable information exchange between any pair or set of tools. The proposal of this thesis an XML-based message format – messages containing information to identify artefacts, and messages defining relationships between pairs of artefacts.

Using messages instead of interfaces allows for looser coupling, enabling integration of a tool without adapting its underlying concepts. It is therefore possible to achieve quick results by extending one tool to send messages about its artefacts, and another tool to process these messages and offer its users the information obtained from them – the first step towards traceability. Using XML as base format allows easy message validation against an XML schema and also offers extensibility to incorporate additional information for tighter tool integration where this is desired.

As proof of concept, two standalone tool prototypes have been developed and then extended to communicate via the suggested message format: a requirements management and a test tracking tool. The set-up shows that traceability links can be established across tool borders and that this provides benefits to the users. With the shared information, for example reports on test coverage or test results of specific requirements are possible.

Keywords: *traceability, software engineering, tool integration, XML messages, decentralized links*

Table of Contents

Table of Figures

Table of Code Listings

1 Introduction

The methods and tools employed to develop software have evolved rapidly in the past decades, from the beginnings of plugging wires to source code on punch-cards to integrated development environments that offer support like code highlighting, code completion assistance and code navigation. Coding directly in binary or in assembly language gave way to procedural, then object-oriented programming languages. As a result, more and more complex programs and applications could be written to meet an ever-increasing demand for automation of menial and calculation tasks, improvements on productivity and intuitive user interfaces.

In parallel, the requirements given by the customer became more and more removed from the implementation – while in the 1970s, developers were handed a software specification where only few degrees of freedom (or uncertainty) remained, requirements also became more user-friendly, in so far as that business people would specify business rules that the system would have to follow and enforce, but not how the system should do that. Designing the structure of the system under development became a task of the development team, shifting the focus from programming itself – which was becoming easier with new tools – to structuring the system so that it would exhibit errors early so they could be easily found, could be modified without massive and therefore risky changes, and be accessible to new developers.

As tools and system complexity increased and the focus of work shifted, the processes followed to develop software from customer requirements also had to adapt. Making processes explicit was the first step towards a long-running and still ongoing discussion about how software development teams should approach a new project. Implicit to these processes was the concept of traceability, being able to follow the trace of a customer input – requirement, change request, defect report or the like – through the stages of analysis, design, implementation and test. Implicit because although any software development process worth this attribution offered guidance on how to get from the input to the output, but did not explicitly state that each intermediate artefact is linked to its predecessor and that at any time, anybody should be able to follow these links and derive information from them why the system has been developed the way it was. This also means information about the decisions and perhaps mistakes made during the project.

Traceability in this sense has relatively recently become a subject of research and discussion. Implementations that support linking and, to a limited extent, navigating between intermediate results (artefacts) do exists, but have not yet evolved to a state where complete project traceability can be realized with ease.

The tools developers have at their disposal will no doubt be further improved and refined, but for complete project traceability, all the tools used throughout the development process have to communicate with each other. While one way to tackle that problem is to only use one massive tool to do everything from requirements analysis to defect tracking, a more realistic way – which is already being followed – is to have a central repository that every tool accesses and stores its data in. The third option, which is the topic of this thesis, is to define a way for existing tools to communicate their data to other tools and have them decide whether to use this information or not.

This third option is especially attractive for development teams that use various inexpensive or open-source tools from different vendors or communities as opposed of a full suite of CASE tools from a single vendor that already offers some traceability features. Even those teams that have a strong affiliation to one vendor might use one or two tools in their development that are not integrated with the rest. Since most closed-source development tools can be extended via an add-on API, these can still be integrated using the third option mentioned above. It is a viable method of tool integration for small development teams, an often overlooked factor in the industry [1], whose productivity losses due to internal communication [2] are not significant enough to justify the investment in a heavy-weight process and the accompanying tool suites: they can keep their tools and their infrastructure, but have access to traceability features that make their projects more transparent and remove some hindrances from the developers' work.

The remainder of this thesis is laid out as follows: In chapter two, the traceability concept is introduced and examined from different angles. Chapter three presents possible applications of traceability in software development, which is followed by a discussion of the poor state of practical usage in chapter four. Chapter five presents the concept at the heart of this thesis: a framework for message-based tool integration, which has been implemented in the prototype described in chapter six. In chapter seven, existing approaches are discussed and compared to the message-based framework.

2 The Traceability Concept

While this thesis will concern itself only with traceability in software life cycle management, a brief digression to other disciplines to achieve a general understanding is helpful before delving into the specialised concept in order to improve one's notion of what can be achieved by implementing traceability:

In the food industry, traceability means that each packaged unit of produce can be tracked back to the farm it came from. In case some kind of contamination – viruses, bacteria, poisons or the like – is discovered at a particular farm, retailers and authorities can remove produce from that farm from circulation, as each unit can be tracked to its origins. A less critical application is proving that a particular cachet such as “organically grown”, “free of genetically engineered organisms” or “fair trade” applies to a unit of produce, if the traceability information includes the standards the producer adheres to and the validating authority. This is nowadays achieved by printing traceability information on the packaging, but may employ RFID chips in the near future.

Industries such as automotive manufacturers employ traceability to minimise cost and effort when a safety issue mandates a callback of, for example, a certain batch of cars. Knowing which parts went into which car enables the manufacturer to call back only those cars that had a malfunctioning part installed.

The same is true for virtually every industry assembling parts that were bought from different vendors, such as notebook computers or mobile phones, where defect batteries have been known to create serious problems, even exploding under certain circumstances.

A different aspect of traceability is the correlation of materials with destructive tests made on materials of the same batch – such tests may be for material strength or chemical composition. This is especially important for differentiating between materials that, while superficially equal, have quite different tolerances for heat, pressure, etc.

Businesses have always had to trace their processes and deliver a report on demand: an archive of bills is a form of presenting business transactions in a traceable way, which is essential as a basis for balancing, which is used for tax calculation and shareholder information. In customer service, processes that take some time must be traceable to be able to inform the customer of the current status of his request.

Summing up the concept, traceability refers to the completeness of the information about every step in a process chain [3] – which means that at any time, it can be determined who did what, when, why and with which entities, so

that the whole process could be exactly repeated. This is an essential part of software engineering, wherefore the Capability Maturity Model Integration for Software Engineering [4] imposes traceability requirements from level 2 on.

2.1 History and Development

Like every other idea in science, the concept of traceability with regards to software development has undergone a development that – in this case – originates in tracing requirements and has evolved to encompass the whole software development lifecycle. In this section, several steps in this development will be outlined along with the rationale behind the extension or change.

2.1.1 Origins

The idea of traceability originates in the early 1970s, a time when the waterfall model (see [5]) of software development was state of the art. The assumption was that a system specification could and had to be complete and stable before development began.

The idea of traceability was first conceived to trace requirements within the software specification; of course, those “requirements” were already technical, framed in terms of input/transformation/output for functional requirements, for example. In [6], a distinction between requirements and specification is made (see below), where these “requirements” are definitely on the specification side.

Especially the military, demanding complex real-time systems, was concerned with verifying that the delivered system did exactly what they needed it to do. Early approaches [7] mandated that in the course of system decomposition, more detailed specifications should trace to their more abstract origins (vertical traceability), and that related specifications on the same level of abstraction should also be linked (horizontal traceability) and always be considered in union to find out discrepancies, conflicts and omissions.

Since this traceability approach is not trivial when faced with a system as complex as ballistic missile guidance, automation (nowadays “tool support”) was a concern from the beginning, in this case automatic verification of the system's specification, mandating a formal specification language. Such a formal language is often not sufficiently understandable for the customers, making the approach somewhat impractical, since there is no way to validate that the formal specifications meet the customer requirements.

2.1.2 Tracing to Design, Code and Test Artefacts

In the 1980s, the concept of traceability was extended to linking “requirements” to design documents, source code and test cases [8]. Again, completeness and consistency were the main concerns, but at this time they were extended beyond the system specification – [9] terms this post-traceability, because it traces the artefacts being created after requirements analysis.

Tracing design documents and code modules to the “requirements” they were supposed to fulfil enabled verification procedures to check that all requirements had been considered in the design and incorporated in the software.

Deriving test cases from requirements and tracing back to them was probably even more important – that is why the system presented in the referenced article is called Requirements-to-Test Tracking System. Traceability helped ensure full test coverage of the specification, enforced the concept of testing against the specification and established the decoupling of testing from coding, enabling the development of test cases before or concurrent to programming – “test before you code” is the key concept of test-driven development, a software development technique popularized almost two decades later.

Unfortunately, the same feature that was so beneficial to testing – tracing to the specification and decoupling it from design an code – is detrimental to the relationship between design and code. Tracing code directly to the specification instead of design (and thereby indirectly to the specification) did nothing to ensure consistency between design and code. Changes introduced at the requirements level would propagate to design models and code, and with some discipline, the code would follow the design changes. Changes in the design – because of technical limitations, for example – would probably be done at code level and never find their way into design documents or models, because of the difficulties involved in propagating changes from design documents to code or vice versa.

Furthermore, tool support in the mid-1980s was still rudimentary: specification, design and test cases were still on paper and therefore not accessible through the database containing traceability links. Source code, although stored in files, was not directly accessible from the traceability tool either. So the traceability tool did nothing but store references to files or hard copy documents and linked their contents (as far as available) with traceability links. Keeping the information stored in the tool up-to-date was the critical overhead task: if it was not done completely and in time, the benefits of the system would deteriorate quickly, because why bother looking up information that cannot be relied upon?

2.1.3 Tracing Sources of Specifications

The next evolutionary step in traceability was to capture where specifications came from. This is where the "Stakeholder" concept, along with high-level (business) goals, is considered. During the process of becoming more customer- and user-centred, problem domain and solution domain became separate entities, and terminology shifted to represent that separation: requirements are part of the problem domain, reflecting customer needs and expectations, while specifications define the system serving as solution to the defined problem [6].

Each part of the system specification must obviously be traced to requirements – if the customer does not request it, the system should not do it. In the other direction, traces ensure that each requirement is incorporated into the system specification. The specification can thus be validated against the requirements, omissions and unjustified "extras" (gold-plating) can be identified. This is assuming, of course, that the requirements are documented in a way that they are completely understood by both customers and analysts, and that they are structured in such a way that pieces of information can be identified and traced – the use case concept was established to provide that kind of interface.

To take this one step further, software requirements are traced back to business goals – strategic or operational – that are supported by these requirements [10] [11]. That aids in understanding the relative importance of requirements, which is important in prioritizing and deciding on tradeoffs between conflicting requirements. Additionally, requirements stability can be assessed by looking at the goals supported, and system design can use this information to plan for changes during or after initial development. The traceability links between artefacts preceding requirements or from requirements to these predecessors are collectively referred to as pre-traceability [12].

2.1.4 Process Traceability

Until now, all traceability links connected only artefacts of software development, which is termed product traceability [13]. Considering the definition of traceability in the IEEE Standard Computer Dictionary,

> *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.*
>
> *The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement*

it satisfies.

product traceability is the predominantly employed form, which is not surprising, since current tool support mostly ends there.

Process traceability extends the concept to include the tasks creating and modifying those artefacts, as well as the people performing the tasks and the resources used to do so, thus extending the concept to the definition found on wikipedia.org:

> *Traceability refers to the completeness of the information about every step in a process chain.*

The challenges and possibilities of process traceability and corresponding tool support will discussed in the following sections.

2.2 Process Model

The Rational Unified Process (RUP), currently an industry standard, will serve as the reference process model for this thesis. The RUP has been assembled on the foundation of best practices, common concepts, activities and rules found in successful software development projects. One of its appeals is the simple and intuitive meta-model (see Figure 2.1).

A *role* is an abstraction for the project members or stakeholders taking a

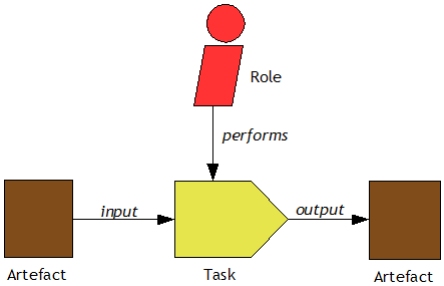

Figure 2.1: Rational Unified Process meta model

particular view or responsibility. A role may be taken by several people, and one person may take on several roles during the project.

A *task* is the smallest piece of work that actually creates a valuable result in the course of the process. This valuable result reflects in the creation and/or modification of one or more output *artefacts*. In order to perform a task properly, one or more input artefacts containing necessary information or old

versions of the future output artefact may be necessary.

Disciplines and Phases

The Rational Unified Process groups related tasks into activities, which compose workflows. Each workflow belongs to a discipline of software development, of which the RUP identifies the following:

- Requirements
- Analysis and Design
- Implementation
- Test
- Project Management
- Configuration and Change Management
- Environment

The first six are core disciplines, named because they directly contribute to creating and deploying the required software. The last three disciplines take a supporting role – without them, a project would not be conceivable. In this thesis, I will try to touch upon each discipline, showing how traceability and its applications enable or enhance the corresponding workflow, and how traceability information about the workflows of each discipline adds value to the project.

While disciplines categorize the project work by grouping related tasks, phases split the project temporally, each one dedicated to a different focus and ending with a defined milestone. Phases according to the Rational Unified Process are:

- Inception, where tasks are focused at understanding the problem
- Elaboration, where the focus lies in finding a workable solution
- Construction, where the elaborated system is built
- Transition, where the product is made available to the customer

Phases do not strongly relate to traceability, although generally, in earlier phases traceability information is rather collected, while in later phases that information is used – to guarantee completeness and correctness, and to improve productivity. It should be noted that the process says little about traceability and how to use it, and that gathering and using the corresponding information is not explicitly part of the process.

Artefact Aggregation

Tasks deal with artefacts at several layers of abstraction. This is necessary to keep the model understandable, because a vast number of small artefacts as

input to a task would obstruct the view of what kind of information must be available to the person performing the activity. Related artefacts are therefore aggregated into more encompassing artefacts, which can of course be part of another artefact of even higher level. Traceability support is required at both coarse- and fine-grained levels [14].

For example, consider the case where a task creates or manipulates a single design class, and a task that reviews the entire design model. It would be misleading to declare the whole design model as input of the first activity, and impractical as well as incomplete to declare all design classes as input for the latter – incomplete because the relations between classes would be missing.

In this case as well as many others, it makes perfect sense to declare that a design class, although an artefact in its own right, is part of the design model, which is also an artefact.

Artefact Attributes

Apart from the content of an artefact – for example, the project vision written down in the vision document – there is often meta-information that should be documented along with the artefact: the responsible person, dates of creation and last change, approval status etcetera.

That kind of information is called artefact attributes; there may be attributes for all artefacts, a specific kind of artefact, such as documents, or a single artefact. These attributes may, of course, vary between projects. It is only important that they are defined and their contents associated with the artefact that they describe.

The attribute examples above relate to traceability: the responsible person should be derivable from the activities that created the artefact; creation and change dates, along with the person creating or changing the artefact, should captured by version control mechanisms, as a full version history of any artefact is part of complete process traceability; the approval status is the result of a decision-making process, which should also be traced as advancement on the agreement dimension (see Traceability Model, below): who approved or disapproved of which version when?

Roles and Activities

Artefacts have to be created, maintained and used to create other artefacts in order to accomplish anything. The act of doing something with an artefact is called “activity” in the process model. Activities can take existing artefacts as input, modify them or result in new output artefacts. For example, when implementing a class, the corresponding design model element is the input and

the source code file is the output.

Activities are of course carried out by people (“workers”), but since the process model cannot know the actual team members by their names, or even the number of people in the project, the concept “role” is established to define a set of criteria and skills required to perform certain activities. Any worker can take on one or more roles in the project, and a role may be taken on by several workers, e.g. “programmer”. Activities are thus, in the process model, carried out by roles.

Type and Instance Levels

From the examples of “design class” and “design model” above, it should be evident that artefacts exist at a type and an instance level. Of course, an artefact type may be instantiated only once in a project, as may be the case with the design model. However, many artefact types, such as “design class” will be instantiated several times, each instance identified by the class's name, for example.

The same is true for activities – an activity type will typically be instantiated several times during a project, once per iteration. Even the relationship of role and worker can be seen as type-instance relationship, but this is not quite accurate, since a worker can take on many roles.

The process model works at the type level, for example the activity “detail use case” modifies the artefact “use case” - both activity and artefact have to be instantiated for each use case in the project. Likewise, a project model can be established as an instantiation of the process model, containing all the artefact and activity instances as well as actual workers as opposed to roles.

Example Workflows

In this section, three simple workflows are presented, which are based on the Rational Unified Process, but simplified for clarity and focus of discussion. Examples are taken from the Requirements, Implementation and Project Management discipline, respectively.

In the example given in Figure 2.2, the system analyst is the only role performing activities; in fact, most of the work laying the foundation for the system's definition is carried out by the system analyst, the details are worked out by the requirements specifier.

The business case is a document of the project management domain describing the commercial value the envisioned product will have for the customer, while the iteration plan “is a time-sequenced set of activities and tasks, with assigned resources, containing task dependencies, for the iteration”. The business case

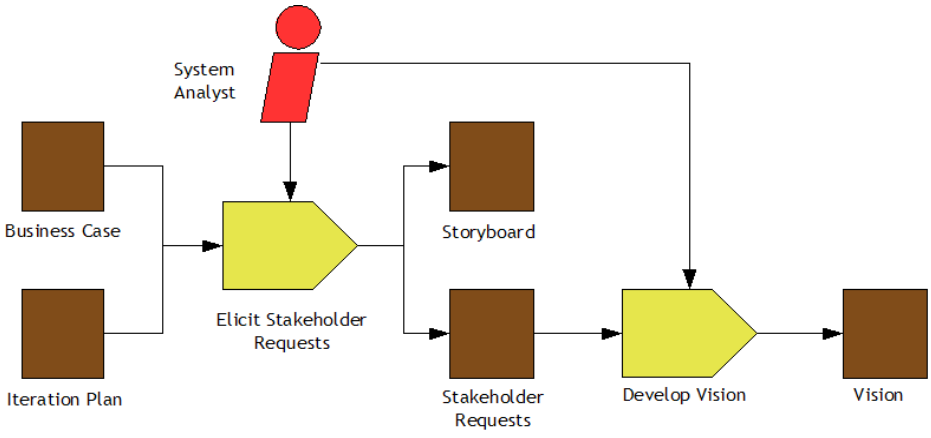

Figure 2.2: Requirements worklow example

helps to find stakeholders and prioritize their requests, while the iteration plan defines the available time and resources for the tasks at hand.

In the course of eliciting requests, various techniques may be applied to gather information, storyboarding being one among them – this method results in storyboard artefacts. Other information, such as interview and workshop results or pending enhancement requests are captured in the stakeholder requests document.

This document is the essential input to the next task, developing the vision. The vision document outlines the envisioned system – its functionality, area of application, expected benefit et etcetera. The vision is an essential artefact, as it provides the focus for the project – the whole development effort should be directed towards fulfilling the vision.

In the second example (Figure 2.3), there are two roles: the implementer who codes parts of the product, and the technical reviewer who looks at the code to ensure that guidelines are observed and best coding practices are applied. The flow of work is rather simple: parts of the design model are implemented, resulting in source code files and corresponding binaries, which are reviewed and tested respectively. The results of both review and developer test are captured in documents, so that they can be used to improve the code and analyse the runtime behaviour.

The interesting fact is that the coded implementation element serves as input to the development task. That is because in following iterations, the element may have to be modified or extended.

The third example (Figure 2.4) is taken from project management and has been chosen because monitoring the project status relies heavily on the fact that information about the ongoing process of software development must be

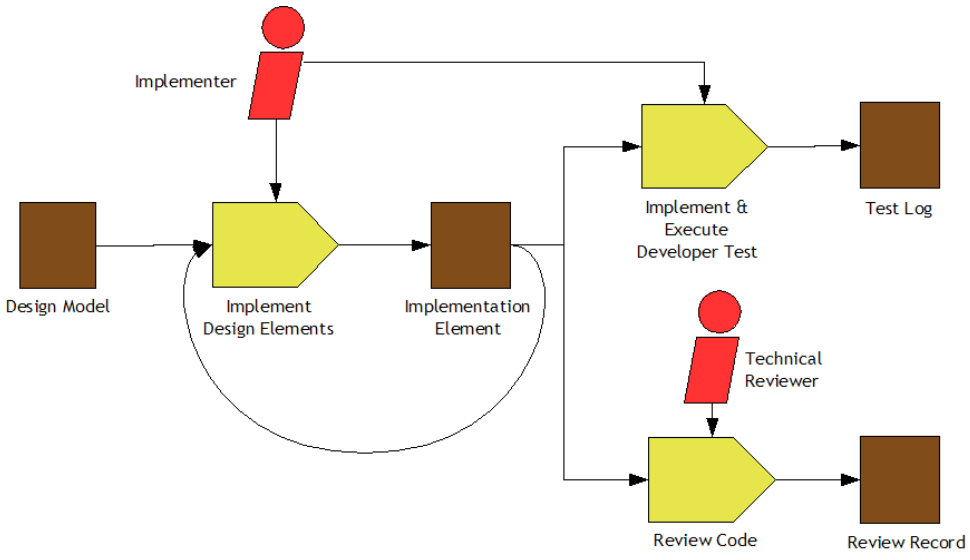

Figure 2.3: Implementation workflow example

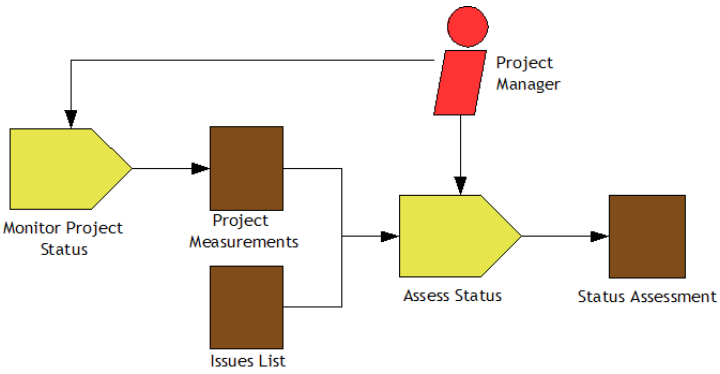

Figure 2.4: Project management workflow example

available. Reliable information about progress cannot be obtained by asking the team "what percentage of the project is done?", but instead by evaluating existing artefacts – e.g. use cases and their review documents, or source code and corresponding test results – and information on time spent on activities versus time planned for these activities. In essence, to effectively monitor a

project's status, traceability information must be available.

The process of assembling the information in a meaningful manner and its result, the project measurements, as well as the conclusions drawn (status assessment) and the decisions based on these conclusions should be traceable themselves – being able to explain how the project's status has been assessed will improve credibility and help justify resource requests or feature cuts if necessary.

2.3 Traceability Model

Since “traceability” is a term to catch all kinds of information on processes and their results at various levels, this section establishes a framework to categorize several aspects of the concept.

2.3.1 Three Dimensions of Traceability

According to [15], requirements management aims at advancement in three dimensions: representation, specification and agreement. This model can be extended to the whole development process.

The **representation** dimension refers to the degree of formalism used to express information. Since source code – the primary human-readable product of a software project – is highly formal, and automation of tasks such as checking for completeness requires a certain level of formalism even in preceding artefacts, advancement along this dimension is critical. Formalism is not restricted to source code: modelling languages such as UML (Unified Modelling Language [16]), possibly enriched with OCL annotations (Object Constraint Language [17]) is another highly formal way to capture information. Even using structured text to express requirements is more formal that unstructured prose, but requirements can also be modelled [18].

For traceability, this dimension means that each formal representation of information can be traced back to a less formal source, ending at the original stakeholder input, which could be an email, telephone call, meeting or hallway discussion.

Specification means the detail to which the problem is understood or the system is specified. In order to build a solution that fits the problem at hand, the problem itself has to be assessed and fully understood – the process of gaining the necessary insights and making that knowledge available to the project team is called requirements engineering. A certain degree of specification must be reached before a solution concept can even be conceived, but a complete specification will almost never be available before the product is finished, software development and requirements engineering running in parallel to fit

the solution to new facets of the problem discovered only because the solution did not solve them.

The finished product specifies the solution completely; traceability along the specification dimension helps to ensure that it fits the problem.

The third dimension, **agreement**, exists because of the multitude of stakeholders typically involved in a software project. There are often many views of what the problem at hand really is, and various interests and concerns that have to be weighed against each other to find a common set of requirements. Within the project, design decisions and tradeoffs have to be made. Advancement along this dimension means that decisions that everybody can live with are made.

In order to avoid re-discussing issues that have been decided upon, or to be able to revisit a decision after learning new facts, the process of coming to a conclusion has to be traceable, with alternatives and their rationales readily available.

2.3.2 Traceability at the Meta, Type and Instance Levels

Traceability links must be established between instances of tasks and artefacts. While the definition of how process elements are related and which rules apply to ensure completeness and correctness resides at the type level as part of the process model, additional precautions must be taken to enforce these rules on instance level.

For example (see Figure 2.3), the “implement design elements” activity receives the “design model” artefact as input, part of which is to be implemented[1]; the activity produces the “implementation element” artefact as its output. Since there is only one design model to be implemented, but several components, each “implement design elements” activity can be traced to the single design model serving as input. However, the “implementation element” artefact does not trace back to each activity, but only the one instance that produced it. This is especially important when we take the person performing the activity into consideration: A role is also a class of people that can perform a task type, the concrete people doing it are instances of that role. Thus, for complete traceability, the people assigned to each task instance must be known to identify those responsible for an artefact instance or change to it.

Consider the following example (illustrated in Figure 2.5): In a project, Alice and Bob take the role of implementers; the project's design reviewer has approved of some parts of the design which are ready for implementation – the

1 Of course, one could define a “design sub-model” artefact to serve as input for this activity, but for simplicity, that refinement is omitted

core business logic and the printing component. The project manager now assigns Alice to do the first and Bob to do the latter; thus “implement design elements” is instantiated twice, Alice is linked to “implement core business logic” and when she creates the component, her code will be linked to that task instance, creating a trace from the output artefact to her. Bob, on the other hand, has nothing to do with the core business logic, although at the type level, his role (“implementer”) is responsible for the task (“implement design elements”) and its resulting artefacts (“implementation element”).

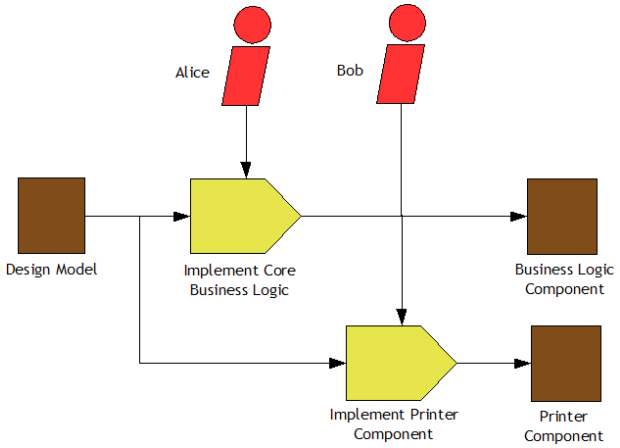

Figure 2.5: Workflow instantiation example

This example also shows an issue with the instance level: At the start of the project, it is unknown how many and which instances of which tasks and artefacts it will create; even the people fulfilling the roles defined in the process model may not be known yet. A process ensuring that complete traceability information is captured must deal with the fact that the extent is a priori unknown, and only becomes clear as the project advances.

2.4 Traceability and Version Control

One of the most fundamental concepts in software engineering, even in small teams, is version control. Keeping a history of changes that artefacts have gone through is one of the foundations of any successful software development team. Although this practice is often applied only to source code files, there are compelling reasons to expand it to all created artefacts:

- Ability to reconstruct the evolution of a particular artefact, e.g. the software architecture document or design model
- Ability to revert to a former version if changes have led to a dead end

- Accountability for specific changes
- Enabling version-specific traceability links

The last point is decisive: to reap full rewards from traceability, all relevant information has to be version aware. Consider this example: test cases are written against a specific version of the specification, and implemented as test scripts that run against a specific build (compiled from specific versions of source files). Obviously, a test script cannot run against a build it cannot interface with, so changes to the design of components may trigger changes in test scripts. Likewise, changes in the specification (a use case, for example) may trigger changes in test cases – in addition to design and thereby to code. Since the test case changed, the implementing test scripts will have to be adapted to implement the new version of the test case.

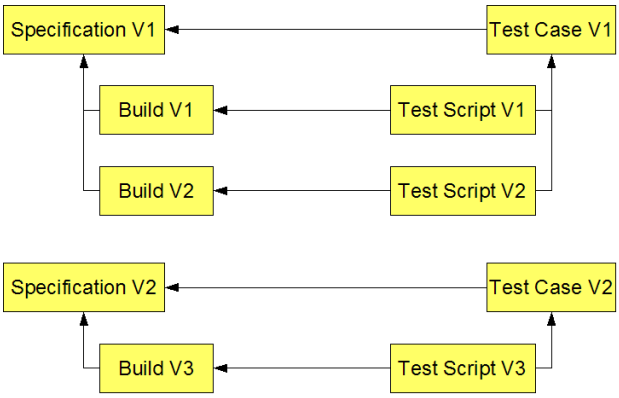

Figure 2.6: Version traceability

There is a difference if the test script is adapted to interface with a new build version or if it is adapted because the underlying test case changed, and that difference can only be reconstructed if test cases are version-controlled. That, in turn, only makes sense if the test case versions can be traced to their corresponding specification versions.

The concept of baselines (see 3.5) encompasses all kinds of artefacts, and so does traceability – both build upon version control.

3 Traceability in the Software Lifecycle

This section is dedicated to the benefits and possibilities of using traceability to the fullest extent in software development, deployment and maintenance. As this is a vast area encompassing many different processes and variants, the discussion is partitioned like the RUP process model, using disciplines as primary and roles as secondary structural concept.

Since the RUP is only concerned with the development and deployment parts of the lifecycle, additional disciplines and roles are introduced to cover maintenance of the productive system.

As it lies in the nature of the concept that traceability is also important at the interfaces between process areas, but the sheer number of such interfaces makes it impractical to create a separate section for each, discussion of the benefits of a particular connection are put with the role on whose work the link has the most impact.

Connections or links are intentionally abstract – there has to be some kind of relationship that a person can follow. This may be as simple as using the same names for design elements and code constructs, or as sophisticated as one-click round-trip engineering, for example. The important point is that traceability must exist; there are myriads of ways to efficiently use that information once it is digitally available.

3.1 Requirements

The requirements discipline was historically the first to be subjected to explicit traceability, because consistency and completeness of requirements are vital to project success and traceability helps to achieve these goals [19][20].

However, as the usage of software spread, the requirements concept changed and became less technical and more oriented towards stakeholders that do not concern themselves with the technical implementation of their wishes and expectations.

In modern software engineering, a requirements analyst uses proven techniques to elicit requirements from these stakeholders and organizes them in a way that architects and designers can use them to build a technical solution to the problem described by these requirements.

3.1.1 Business Analyst

For most cases of software development, there is a sound reason behind the project: a business opportunity, research need or regulations conformance issue. There are high-level goals that an organization is trying to further using the

newly developed, adapted or extended software that is to be the result of the project.

The business analyst is responsible for making sure that the requirements for the project meet the overall organizational goals, keeping the various stakeholders' wishes in line and balancing the contradictory interests of the concerned stakeholders.

When a project is conceived, the business analyst finds the concerned parties within the organization and elicits their wishes and expectations towards the project. These include high-level organisational goals, management view on the problem, the business processes to be affected by the project and the practical issues of those actually performing these processes. The project vision is established from a number of needs to remedy current issues, for example with efficiency, flexibility or traceability of current business processes.

Traceability is achieved when all wishes and expectations are explicitly stated and linked to their origins (stakeholders) as well as to the organisational goals with which they correlate or conflict. Furthermore, decisions concerning the weighing of conflicting wishes against each other have to be documented, including the decision-maker and a rationale for the decision [10].

The resulting project vision is then a solid set of compromises or, ideally, a consensus among all organisational stakeholders, the process of coming to this compromise being well-documented and therefore safe from being re-discussed at large during the project. When additional issues arise, they can be included into the collection of wishes and concerns, matched against existing ones and decisions be re-evaluated on a sound basis.

3.1.2 System Analyst

While the business analyst's job is to identify the stakeholders and capture the a vision of the project, the system analyst is responsible for eliciting more detailed requirements with regards to function, system environment and quality – requirements are to answer the questions of what the system is going to do and under which circumstances it is going to operate, who will use it for which tasks and what the expected quality characteristics are – which means finding measurable quantities for performance, reliability, maintainability and similar qualities.

The result of the requirements analyst's efforts is a system specification that is at the same time a solution to the business problem and an implementation problem to be solved by the development team. That means that a system that behaves as specified adequately fulfils the wishes and expectations captured in the vision while leaving implementation details to the developers as much as

possible. An analysis model showing the problem domain's concepts and their relationships is part of that specification.

To make sure that the specification correlates with the vision, specification elements such as user interface prototypes, technical interfaces, applying regulations, quality figures and the software life cycle projection have to be traceable to elicited requirements, which in turn have to be justified by referring, directly or indirectly, to the relevant parts of the vision.

Furthermore, the process of requirements elicitation, analysis and specification should be documented, so that it is clear whose input led to which conclusions, which decisions have been made, which alternatives have been considered and why they have been rejected. As with the vision, this helps to later defend the system specification against time-intensive discussions, since the rationale behind the specification is available at any time.

3.2 Analysis and Design

In this discipline, the goal is to find a system architecture and design model that describes an adequate solution to the problem posed in form of the specification: the modelled system will, when implemented, behave as the specification mandates.

Traceability at this interface is non-trivial: a system architecture may be a solid base for a system behaving as specified, but how should an architectural model be linked to a set of specifications? The same is true for the design model parts that do not concern themselves with user interface implementation. In this section, approaches for tackling these issues are discussed.

3.2.1 Software Architect

A system architecture is a set of hardware and software components and their interoperation. The architect is responsible for establishing such a set within the bounds of the specification, so that the system will perform its functions under the specified quality characteristics. Where several options are available, the architect has to justify his decision; the process of decision-making (listing alternatives, giving a rationale for the chosen option) should be documented.

When working from a set of use cases, [21] suggests determining the architecturally significant ones in order to guide software architecture. For each candidate architecture, a proof of concept shows how the specified features and quality characteristics are supported by the architecture; this may be a document, a reference to the architecture of an existing similar system, or an architectural prototype that allows testing some characteristics – especially performance – in the system environment.

Traceability is achieved when a proof of concept shows that the architecturally significant use cases are supported, required quality characteristics can be achieved with this candidate architecture, and applying regulations and interface requirements are adhered to. Furthermore, other candidates and why the chosen one was selected can help fend off later discussions, as mentioned in the requirements section.

3.2.2 Designer

Software design is the process of describing the software to be implemented so that the task of programming it can be done in parallel without much need for synchronization. This means that a complete software design consists of a modular break-up of the software and interface specifications for these modules.

The design fleshes out the details where the architecture provides the framework; traceability is achieved when classes or groups of classes are linked to the functional specification they implement and to the qualitative specifications that drove or limited the design; to enhance the latter point, important design decisions should again be documented with a rationale, perhaps even with considered alternative designs.

Design elements will depend upon each other and the persistence solution – accessible class members and the database schema define the interfaces. These dependencies have to be documented so that pending changes to an interface can be traced to the users of that interface, and all clients using that interface can be considered before making the change. Apart from preventing changes that would break the system, extensions to the interface can be considered for all clients at once instead of adding elements for one specific client at a time, which often leads to bloated interfaces.

Within the design model, elements such as classes, methods and properties are often contained within other elements. These aggregation relationships form several levels of abstraction, which can be used to find the desired level of traceability: in the most abstract case, the design model as a whole is linked to each use case, which would not be useful; in the most detailed case, only (public) methods and properties that participate in a use case are linked to it, which would contain maximum information, but would be cumbersome to establish and to maintain. The appropriate solution depends on the project's nature and traceability requirements.

3.2.3 Database Designer

Data persistence is an essential feature of most software systems. For efficient use of persistent storage, appropriate data organisation is vital – this is the

responsibility of the database designer. The name may be misleading, as not all persistence solutions are database systems: formatted text files may be used for storing configurations and XML can, to a limited extend, replace conventional databases. Pictures and streaming media (audio, video) have their own data formats, as well as typical “office” documents: formatted text, calculation tables and presentations.

However, if the system will not use a pre-defined persistence solution, the database designer is responsible for defining the schema or format. This design has to provide efficient access to stored data to the system, which means that its organisation has to reflect the problem domain's concepts and their relationships, as identified by the system analyst.

Tracing tables and columns to the analysis model elements they represent ensures completeness of the design, as left-out elements can be found automatically. Correctness, meaning that relationships and attributes are properly reflected in the design, can be easily reviewed using this traceability information.

3.2.4 User Interface Designer

This role is responsible for defining how the user can interact with the system. This includes the system output to screen, printer and speakers as well as the possible input devices, often limited to keyboard and mouse.

For most applications, designing the user interface means assembling forms from reusable and well-known elements in a way that provides all necessary information and functions without obscuring them with the sheer amount of superfluous ones.

Use cases should be the driving force behind the design – everything a user needs to perform his task must be available to him. This is also the focal point for traceability: a form may serve to realize one or more use cases, so the information on that form should be limited to what is necessary for the user to perform that use case. Since use cases contain information about user input and system output in each step, user interface elements can be linked to the corresponding step.

This level of detail allows automated reviews of forms regarding completeness (each step in the use case, or even each piece of input and output, is covered) and preciseness (no superfluous elements are on the form).

Tracing forms to use cases aids in manual reviews of the user interface, and should be the minimum of traceability information that is captured.

3.3 Implementation

This is where system is actually built: programming the system logic, creating databases, drawing the user interface etcetera. Traditionally, this is also where problems with design and/or architecture are mostly discovered, leading to re-design in code that is often not documented or to hardly maintainable workarounds ("hacks"). Formal software designs are often abandoned because of too many design changes in code, which makes progress monitoring and change management difficult and is related to significant schedule slippages [22].

This section will show how traceability can improve communication among programmers, maintain consistency with the design model and contribute to quality assurance.

3.3.1 Integrator

This role is responsible for planning and executing the integration of implementation elements – building the system from the written components. This means that the integrator has to know which versions of those components are current and work together, as well as the dependencies to external components. The introduction of versions to the traceability model complicates matters significantly, as discussed in Configuration and Change Management.

Leaving the versioning problem aside, traceability is achieved when each software component is traced to the design model elements it implements and to the other components that it depends upon. Having this information available, the integrator can more easily detect which component is responsible for occurring build problems.

3.3.2 Implementer

The implementer's job is to write code according to the design model, building a software component. Linking each implemented module or class to the corresponding design element is essential for propagating change from the design model to the code.

There is more to implementation than just coding a design: some design decisions (those not concerning interfaces) are made at code-level, and algorithms as well as internal data structures are up to the implementer. As with all decisions, the implementer should document the non-obvious ones; additionally, references to the sources of the used algorithms and data structures should be given so that it can be easily determined if there is a proven algorithm behind the implementation, if the algorithm is appropriate and if it has been correctly implemented.

Implementers are also responsible for unit-testing their code, which encompasses writing test cases and executing them, possibly as a test script or program. A link between these test cases and the units they were written for helps to find the relevant test cases for each test run.

Code reviews are a recommended practice in software development contributing significantly to software quality. Making the review process traceable means documenting the reviewed code, the implementer responsible for it, the reviewer(s), date/time of the review, and the suggested changes.

Each change to the source code leading to a new version should have a traceable rationale behind it. Code reviews, or the suggested changes, are one source of rationale. Another one are defects; defect tickets have to be traceable to a particular version of a particular code module, so that the responsible implementer can be informed and guided to the code causing the problem. With traceability information readily available, the time spent on searching and looking at source code – activities accounting for 40% of the time spent by software developers [23] – could be significantly reduced.

The third source of change is a change request, as discussed below.

3.4 Testing

Testing is done to assess software quality – testing for correct functionality is the main part of the effort, but the system has to be tested against the non-functional specification, too – performance is the easiest of those figures to test.

Tests are often layered as unit tests, done by the implementers themselves, component tests and system (integration) tests, for which special roles exist.

Testing is only efficient if the tests cover all requirements – which means every possible path of every use case has to be considered for functional testing (positive test cases), and deliberate illegal derivations from those paths as well (negative test cases) – and the test results can be used to locate the defect in the source code. This section discusses how each test role contributes to those goals by providing traceability information, or uses that information for her work.

3.4.1 Test Analyst

This role is responsible for defining the test cases and evaluating test results. Since test cases should be derive from the specification in order to be complete and adequate, the analyst uses information such as use cases or quality requirements to create coarse versions of test cases.

After the testers have detailed or implemented the test cases and run the tests, the test analyst uses their test logs to assess overall product quality, identify and

capture sources of non-conform behaviour (defects) and make proposals for corrective action.

Traceability means that each test case is based on the product specification, linking the test case with the corresponding use case (variant) or non-functional requirement. Furthermore, defects should not only be associated with test cases, but – after analysis – link to the defective code module.

The first kind of traceability enables project members to assess test coverage, as specification parts without linked test cases are untested. The second kind of links allows implementers to efficiently look for defective code, while the architect and the project manager can derive problem areas from the number of defects associated with a particular code module.

3.4.2 Tester

A tester details the test cases defined by the test analyst – either by writing down each step in the test procedure or implementing a script – and then executes the tests on the current version of the software. When necessary, the tester also creates test data with which to run the tests. The outcome of each test case is recorded in a test log for later analysis.

When detailing abstract test cases, the tester may use information from the user interface specification, which should be traceable. Furthermore, used test data should be accessible through the test case, so that it can be re-run at any time. This is important for efficient regression testing of newer versions of the tested software.

3.5 Configuration and Change Management

Configuration Management is concerned first and foremost with keeping track of the evolving versions of products and collecting matching ones in a baseline – a snapshot of the project's achievements at a certain point. Change Management, on the other hand, refers to the process of suggesting, evaluating and denying or introducing changes to current versions of products in a controlled way.

Versioning of products represents a kind of traceability – different product states are linked to each other. Since evolutionary steps, requested changes or defects to fix drive the development of a new version, associating these driving forces with the new version ensures that the rationale for changing the product is documented – and thus the change and the effort put into it justified.

A change request is a process element capturing somebody's wish to change a previously agreed-on product. The process of introducing the change into the

system under development begins at the highest possible level of abstraction; through the traceability links described in the previous sections, the change control manager can determine the involved products and – with the help of the responsible people – estimate the necessary effort to implement the change. An authorized stakeholder or change control board can then decide whether to implement the change or refuse to; this decision process should be documented like any other.

When implementing a change, the new versions of changed products should refer to the change request(s) implemented in the new version, so that the configuration manager knows which versions to include in a baseline and the change control manager can track the progress of implementing the change.

3.6 Project Management

Each team but the smallest need a leader who assigns and schedules work, acquires necessary resources and makes decisions when necessary. A project manager is often also responsible for keeping the organisation's management informed about the project's status.

When planning and scheduling work, a project manager creates instances of activities an assigns them to people to carry them out. Traceability is achieved when the project plan contains all planned tasks and those carried out so far, if the persons performing the activities are known and the results of these activities are referenced. Of course, that means that the project manager has to synchronize his plans with reality, adapting to derivations from the plan.

Status assessment is eased by having traceability implemented in a project [24]. Since concrete results are always linked to more abstract ones, lack of links leading to the abstract products shows where there is still work to do. Change requests are associated with the highest-level changed product(s), so their implementation can be monitored by following traces to the more concrete results and checking if the current version has implemented the change. Defects and their state of fixing are also tracked and therefore subject to monitoring; their links to the concerned code modules can be evaluated to find problem areas and serve as guide for management action.

3.7 Customer Support and Product Maintenance

Product maintenance encompasses capturing and collecting defects, developing fixed versions and delivering the software update. Furthermore, keeping the system running in a production environment may necessitate periodic or emergency data maintenance, such as backup/restore, purging unused data or manually correcting invalid data.

As with defects detected by the test efforts during development, reported problems have to be logged and analysed in order to determine if it is indeed a defect or a change request, which parts of the source code are concerned and which effort will be needed to change the code. Traceability information captured during development is essential to these tasks.

Collected defects and approved change requests can then be passed on to developers in order to create a new version of the software and an update package. In the meantime, users can be informed about known defects and workarounds.

When directly manipulating data the software uses, information about the meaning of data values is necessary. Furthermore, knowing which modules read or write data items that have been found corrupted or superfluous helps to track down the error and enables developers to check all parts of the code that use these data items for analogous errors or incompatibilities with the planned fix. Traceability between the software design model and the data model provides the necessary information.

Customer support includes an end user manual or on-line help that is synchronized with the software version the user has. In order to achieve this, each part of the documentation must be linked to the corresponding product – use cases, user interface specification, data model elements or even code modules. This allows the technical writer to track changes made to the documented products and adapt the documentation accordingly.

4 Deficiencies in Practical Traceability Usage

Although the concept of traceability is not a new one, implementation in development projects an usage in the maintenance process is mostly selective at best. This section is concerned with the deficiencies in traceability usage, proper tool support, and their consequences, with a focus on small software engineering teams.

4.1 Pervasiveness of the Traceability Concept

While there is little data on the grade of traceability usage in the software development industry, there are some studies that can serve as indicators of low pervasiveness, which back up anecdotal evidence that traceability is widely disregarded.

One factor, investigated by [25] is that documentation is rarely updated – with the exception of testing documents, which is usually updated within days. Developers mostly agree that functionality marking documents as "potentially requiring updates" would be useful. This offers two conclusions: either documentation does not carry a benefit for the developers, but is only done for compliance with a mandatory process, or the effort required to keep all that documentation updated is greater than the perceived benefits of up-to-date documentation. If software projects had usable traceability information available, updating documents would mean considerably less effort, which could lead to more immediate updates, improving communication.

Another study, described in [26], found that out of nine companies participating in a requirements process improvement experiment, eight had a "weak" and one an "average" rating in the "management" section in the initial assessment. While scores in the other seven sections were not stellar either, this was the worst overall section score.

In a study about controlling software project risks [27], four of the fourteen risks addressed more or less relate to traceability:

- "unclear or misunderstood scope/objective" marks a failure to promote understanding and trace the agreement process.
- "lack of effective project management methodology" denotes to lack of control over the project, a problem that may be alleviated by traceability as pointed out in 3.6.
- "developing the wrong functions" refers to the problem that actual development does not relate properly user or customer requests. Implementing traceability would ensure that project members have

access to user requests (requirements, change requests) through links from their work products, so that misunderstandings can be easily avoided.

- "gold plating" is a corollary to the previous point, also avoidable through traceability: since a developer would not find a matching user request to link his extra work to, project management can find out gold plating easily. Accuracy of traceability links could be ensured by reviews.

In his IEEE Spectrum article "Why software fails" [28], Robert Charette identifies twelve reasons for project failure, six have a relation to lack of traceability:

- "Badly defined system requirements" may be caused by failing to promote agreement (as above), but also by not providing a context for requirements: stakeholder goals and business objectives.
- "Poor reporting of the project's status" may refer to expressiveness of the report form or accuracy of project data. Traceability helps address the second cause, as outlined in 3.6.
- "Poor communication among customers, developers and users" is not directly addressed by implementing traceability, but can be made obvious early in the project, so that project management can address the situation. Traceability would help to discover lack of request documentation, since there would be not requirements or change requests to link work products to, and would also point to ambiguous requests when project members try to find a rationale for their work to link to.
- "Inability to handle the project's complexity" is often rooted in communication deficits, which can be addressed by implementing traceability to keep a chain of sources for each work product, to keep documentation up-to-date, and to enforce clear responsibilities for each team member.
- "Sloppy development practices" may be result of a lack of competence or lack of motivation to adhere to best practices. The latter is usually caused by organizational or project culture, where "nobody cares" or there is a sense of time pressure leading to abandonment of best practices. Either way, introducing traceability adds accountability; combined with improved monitoring possibilities, this would alleviate the problem.
- Finally, "poor project management" may be the result of little information about how the project is doing, so that appropriate measures cannot be taken. Lack of traceability contributes to uncertainty about the

true project status, as reported figures may or may not be accurate.

These studies support the anecdotal evidence that traceability is not a widely used concept, if the assumptions about what effects traceability has on software development projects hold. A fair amount of problems a software project may face can be directly or indirectly addressed by implementing comprehensive process traceability. However, that may not the only way out of the continuing crisis first announced in the Standish Group Chaos Report 1994 [29].

4.2 An Alternative Approach: Extreme Programming

A great part of software development is done in small teams (up to twelve people), which can communicate and work together more efficiently than a whole division of developers [2]. While the latter needs rigid organization and formalized communication channels in order to deliver a functional product, a small team can interact face-to-face without too much wasted time.

Kent Beck [30] has put together a process for small teams that leverages that advantage and relies on direct, open communication as well as some practices carried to an extreme, and called it “extreme programming”. Practitioners claim increased productivity, less defects and more customer satisfaction since embracing the process.

At first sight, a lean process like XP makes collecting traceability information seem like overhead, not adding much value to the project because information takes different routes than documents.

4.2.1 Extreme Programming Condensed

To be able to discuss that approach, a brief introduction to its core features is in order. The main concept in extreme programming (XP) is the story: a story is a customer request, framed as prose explanation of envisioned user experience of a certain system functionality. Stories can vary wildly in scope, so that a simple change to a single dialogue is as much a story as a full-fledged use case. Stories may be broken up in tasks so that estimation will be more accurate and implementation can be done within a short iteration.

Planning cycles are quarterly and weekly. During the quarterly cycle, a product strategy is devised and a focus for the following quarter is established. Weekly planning lays out the work done by each team member during that week's iteration. Essential to planning is that the customer or the product manager (as customer representative) is responsible for picking the stories to be implemented from the pool of written stories, so that the most valuable stories are implemented first.

An important aspect of XP is test-first programming: for each change, first write a failing test that will succeed when the change has been made – this is done on unit as well as acceptance test levels. Test automation enables constant regression testing, so that the team can be confident of the quality of the software they write.

Pair programming in changing pairs has several effects: first, since two minds work better at thinking of possible test cases or catching mistakes than one, software quality is enhanced and implementation speed is increased so much that two programmers working independently tend to be slower than as pair. Second, each developer on the team gets to see and program on many parts of the product, mitigating the risk of an essential team member leaving and a piece of software remaining without anybody knowing how to maintain or change it.

When the customer (or representative) adds a story to the pool of requests, it is broken down into tasks and the required effort is estimated on a task level. Each week, the customer picks a number of stories with an estimated sum of effort up to the team's capabilities for a week. Tasks are preferably given to those who estimated them, who start out by writing acceptance tests. After that, they implement the requested functionality by writing a unit test before each change. Continuous integration (several builds per day) ensures that the changes do not break the system apart and keeps the cost of integration low. When all acceptance level tests pass, the changes are ready for deployment.

Traceability is not really a considered issue – Kent Beck [30] states that traceability is "built into XP", it only has to be recorded:

> *Traceability, the ability to link what has changed in a system to why it changed, is built into XP, although the information isn't routinely recorded. The only change to implement traceability is to make a physical record of this information. If I am changing this line of code, it is because I wrote that test which is part of that system-level test which came from that story which was scheduled May 24 and was ready to deploy on May 28.*

Of course, traceability is "built into" every process worth being called a process. With a plethora of artefact types, it is only more difficult to sum it up in a one-paragraph sentence. Traceability in XP is a non-issue in Kent Beck's Book, being only mentioned in the section about using XP for safety-critical systems, where physical records of traceability are required.

4.2.2 Traceability versus XP

Since practitioners of XP claim a radical increase in productivity as well as project transparency, the question in this section is: does traceability provide

significant benefits in a small team using XP or a similar light-weight process? Or, in other words, is using a lightweight process an alternative to implementing traceability?

First and foremost, while the “story” concept is a single interface to the customer, it really is either a requirement or a change request – it might even be a defect report (although practitioners claim extremely low defect rates, due to extensive automated testing). When a story is a change request, it overrides or extends an existing story that has already been implemented.

The first step in implementing such a change request should be to find and change the affected acceptance tests, but without traceability, this may prove cumbersome, depending on the number and organisation of the project's current test cases. The alternative is to write new acceptance tests and to revisit those test cases that fail due to the change to determine whether they are outdated or signal that the change has unintentionally broken functionality. This means there is waste creating test scripts that could potentially be more easily adapted from existing ones, and waste in analysing a test script for its continued validity.

The same is true for unit test cases – finding those that belong to a specific acceptance test is quite impossible without proper traceability. Again, writing new ones and dropping outdated test cases after inspection is a viable solution, but a wasteful one. A change request should propagate through the affected process artefacts – nobody would consider rewriting a code module because of a change in its requirements (unless perhaps it is radical, but even then XP suggests refactoring in small steps), so why do that to the test scripts?

Story estimations are also problematic without traceability. Even though project members learn about many aspects of the system under development due to the pair programming practice, and even though at least one person should know much about the part a specific story changes, estimations based on a person's mental model of the system are less accurate than they can be using a proper change impact analysis. Even knowing exactly how many unit tests have to change can provide insights to the extent of the change. Of course, this requires readily available traceability information and supporting tools.

4.2.3 Conclusion

Extreme Programming, in contrast to the Rational Unified Process, produces little to no formal documentation, relying instead on direct open communication and self-documenting code as backbones of a project's information flow. This reduces the apparent need for traceability, as there are fewer artefacts through which a change has to be propagated.

However, given that there are still at least four levels of artefacts remaining – story, acceptance test, unit test, code – traceability can still improve working experience and productivity by helping to eliminate waste. [31] also suggests that agile processes in general need tool-supported traceability as well.

4.3 Reasons for Lack of Traceability

Collaborative software development is mostly a matter of communication: communication between customers and development team, and between developers. As Frederick Brooks describes in [2], cost of communication counters the effect of partitioning work and may even outgrow the benefits thereof. Making communication more efficient is therefore a key issue in software development – traceability is a concept to achieve that goal.

Since the benefits are substantial in theory, the question is why traceability is not widely implemented in software development. Possible inhibitors are named in [32]: traceability as a mandate, only done for standard compliance; using ad-hoc practices with no clear strategy; tool incompatibility; use of traceability for performance appraisal instead of quality assurance. This section will outline some other major obstacles and shortcomings hindering adoption.

4.3.1 Traceability as Investment

When developing new software from scratch, collecting traceability information about the artefacts as they are created requires effort. Depending on the supporting tools, that effort may constitute a substantial barrier to those who have to actually record that information, resulting in sporadic, low quality traceability information that does not provide enough benefits to justify recording it, starting a downward spiral that leads to an abandonment of the traceability concept, at least from the developer's point of view, who do only what is mandated by management.

Even if during the initial project phases, when the team is still confident and motivated, traceability information is recorded, it may sooner or late be neglected because of time pressure building up. When teams are put under pressure, they often regress to modes of work that they are used to, even if the past has proven that they do not help the situation [30].

The fundamental flaw is that the effort required to record traceability information is regarded as a burden instead of an investment. Benefits do not become obvious until change requests have to be serviced or acceptance test failures have to be tracked down, at least to the developers. (Project management may reap early rewards even during elaboration, but the added transparency may make team members anxious – see below).

The investment to be made is even more daunting when an existing software product is expanded or undergoes maintenance work. The effort required to analyse enough of the product to do your job, and to actually do it, is probably less than would be required to retro-fit traceability information throughout the system. Even if those responsible for maintaining systems would gladly spend that kind of effort to make their lives easier in the long run, management often demands immediate results at the lowest possible cost, failing to see the long-term benefits. As with refactoring, a possible approach is to add traceability information where you change something, gradually adding traceability information that will help you when the next change is due in that part of the system.

If developers are in a mindset that their job is creating models and writing code, while quality assurance and change management are somebody else's problem, when shipping the product is the only goal because maintenance is somebody else's problem, it may prove challenging to implement traceability during development. A prerequisite for successful adoption is therefore that those who spend the effort recording traceability information actually reap the benefits – or face the consequences of neglecting to record usable information.

4.3.2 Fear of Transparency

While honesty and sincerity are generally regarded as virtues, in business reality it may actually be disadvantageous to report the true status of a running project to senior management or customers. Especially if there is a pre-determined project duration or maximum cost, or when re-scheduling and adjustment of estimations are regarded as failures rather than natural consequences of learning more about the project's scope and complexity, optimism bordering on delusion is the norm.

While delusionally optimistic projects to not meet a pre-set deadline or budget either, motivation is less of a problem in the earlier phases of the project. If a team learns early on that it cannot possibly achieve unchangeable objectives, how can it keep up motivation? When pointing out problems before they become glaringly obvious is regarded as “can't do” attitude, why bother trying to discover them as soon as possible?

In such an environment, the transparency brought to the project by tracing artefacts and their relationships as they are created or modified – project management can determine exactly how many work items have been completed are how many are still open, as well as the current estimate for each item – can scare developers and project management alike. Individual developers sometimes take longer than the estimated time to finish a work item, possibly

delaying dependent tasks, while project management has to deal with discoveries of tasks that have not been thought of before, with unexpected dependencies and other issues that delay the project as a whole. If and when the blame for a late-running and/or over-budget project is assigned, traceability information could provide a basis for those assigning the blame.

When a project discovers that a full-scale implementation of all requirements is way over the original estimate, it basically has two choices: telling the customer about it and trying to negotiate a new scope or budget, risking project cancellation, or covering up that discovery as long as possible in order to get the customer committed by a large investment so that he will not back out when the system is not ready in time and budget, but will spend additional money to finish it. Short-term business logic would suggest the latter; risking the loss of a contract can only be beneficial in the long run, when you build a reputation of honesty that attracts more customers. While traceability information may be kept internal to hide the true project status from a customer, lying outright is always more difficult (and possibly a legal problem) than being overly optimistic because one does not have accurate measurements.

Because traceability entails transparency, it can be threatening to those working in an unhealthy environment. Where organisational culture and policies reward delusional optimism and “crunch time” efforts instead of honest and regularly adjusted estimates and schedules, traceability has no place.

4.3.3 Lack of Tool Support

While traceability can theoretically be achieved even by maintaining traceability matrices on paper, proper tool support for establishing, maintaining and, for maximum effect, navigating traceability links is vital to an efficient use of the concept [33][34]. As laid out above, traceability ought to save the developer some work by providing information about and access to artefacts that otherwise would have to be found out by analysing existent ones.

Source code files provide an opportunity for storing traceability information: for each code module or class, source code comments could point to unit tests, design models or documents and the like. An advantage of that storage location is that it is edited by the same tools that developers use for writing code, so there is no switching of tools or media involved in maintaining traceability links. Of course, source code cannot be the only location for traceability links – there are several documents, models and possibly even code maintained in other development environments that need to be traced. While any word processor or modelling tool provides means to arbitrarily insert information and thereby offers an option for recording traceability links, spreading information like this

in a non-uniform way makes accessing that information cumbersome and prevents programmatic access to traceability links.

Because of these drawbacks of text-based traceability information, development tools often support linking artefacts produced and maintained by these tools to each other. Since information kept in such tools tends to be structured instead of prose text, in theory pieces of information could be linked at any level of detail. However, even if a tool actually allowed arbitrary links, it would still be limited to the set of artefacts processed with that tool, forcing developers to use the above text-based approach for traceability links crossing tool borders.

Integrated tool suites or tool families built around a central repository tackle that problem by providing a set of tools that are designed or adapted to allow traceability links between artefacts processed in different tools. For example, the requirements management tool could allow links to use cases modelled in the visual modelling tool. However, integration is often limited in the ways of only allowing specific kinds of artefacts to be linked. In addition, relying on vendor-specific tool integration forces the team to bind itself to a specific vendor and his tools, without being able to use the best tool for each project and purpose, because partially deploying such a suite would leave holes in traceability coverage that would have to be bridged with text-based links. Furthermore, obtaining licenses for well-integrated commercial suites is prohibitively expensive for many small software development teams; unfortunately, open source counterparts are not available (see below).

Proper tool support for traceability covering all aspects of a software development project is not easy to obtain, and is most certainly a costly effort, binding the team to one software vendor and forcing a switch to his tools. These reasons contribute decisively to the lack of prevalence of the traceability concept.

5 Proposal: Message-based Tool Integration

As discussed in the previous sections, existing development tools suffer from serious drawbacks in their support of traceability for small software engineering teams: commercial suites are too expensive to allow a return on investment within a reasonable amount of time; the only serious free tool integration effort has failed and the open source community has not managed to establish a project that has produced anything remotely comparable to an integrated tool suite.

This thesis does not aim to define requirements or outline an architecture for such a suite, but instead offers a means of communication that will allow open source projects that are developing and maintaining software engineering tools to gradually integrate with each other, relying on a common standard of exchanging traceability information. As a first step towards this, messages for establishing and maintaining artefact traceability are defined and their use demonstrated in a prototypical implementation. Building on these messages, expansions towards tighter integration on the one hand, and full process traceability on the other hand are briefly explored.

5.1 Concept

OPHELIA defined interfaces for each process area, forcing tool developers willing to integrate to adapt to one or more of these interfaces, a fact probably contributing to OPHELIA's failure. In contrast, the suggested communication standard relies upon the building blocks of the process meta-model (artefacts only in the initial step), as well as the possible links between them, to structure communication, thus being generic and universal. Messages contain a minimal set of required information, but are open for additional information to allow tight integration between tools that need to work together closely.

Actual communication is about events related to project products at the instance level. Messages about operations on these products and their traceability links are exchanged between the tools used in the project in order to maintain a common set of existing products in all tools. This means that whenever, for example, an artefact is created, the tool which was used to create it sends an artefact creation message to all other tools, which contains information about the artefact that allows other tools to identify and classify the new artefact, thus being able to reference it within a traceability relationship.

Messages are distributed by a pre-agreed message bus to which all tools have access. The technical implementation of the message bus may vary, from smtp/pop email or a network file share to a dedicated web service or message-oriented middleware. What is important are the facts that every tool has access

to the message bus and that the message structure is standardized but extensible, so that tools can rely on certain information and may choose to provide and/or process additional pieces of information.

Traceability links between project elements are typed in order to provide automatically processable information on the nature of the elements' relationship. While these types ought to be specific enough to convey the relationship semantics, they also need to be generic enough to be able to classify all conceivable links. The proposed solution to this is to declare a limited set of generic types and allowing tools to make up their own specific types. The rationale behind that is that while there are few types of relationships that behave differently – we could look at UML for a list of relationship types, for example – there is often a need to convey more information than "this artefact depends on that artefact".

This need can be met by free-form relationship types – the specific type name can express the true meaning of a traceability link. There is little danger of having an exploding amount of relationship types, because artefacts of any given type will typically be maintained in one tool exclusively, thus relationship types between two artefact types will only be negotiated between two tools. Taking into consideration that tools tend to cover a specific problem area related to one discipline and the fact that disciplines build upon the work done in previous ones within an iteration, there is a general tendency towards a directed dependency of one tool's artefacts upon another tool's artefacts. Therefore, links will probably be established in one direction between any two tools, so that traceability types are declared by the dependent tool only.

More likely than an explosion of link types is a collision of specific type names. Since a traceability link type can be defined as a triplet of source artefact type, target artefact type and relationship type, this is not a problem either – typically, there will be only one relationship type between any two artefact types. As outlined above, only the tool maintaining a specific artefact type will declare traceability link types with that artefact type as source type. For each source artefact type, the tool declaring traceability link types can use the same type name for different target artefact types, but must differentiate link types to the same target type by relationship type name.

5.1.1 Messages Instead of Interfaces

An essential part of the concept is that tools are not to be forced into implementing an application programming interface in order to participate in traceability communication, but instead send and receive standardized messages. While in both cases the tool has to learn a way of communicating with

others in a standardized way, messaging is a looser way of coupling tools than an API implementation:

- A tool only has to embrace the concept of artefacts (at a later point, workers and activities) as fundamental process elements, the basic operations on them and a small set of generic relationship types between process elements, in order to communicate. With an API, a tool would have to commit to performing specific operations in addition to embracing the concepts set forth by the API's designer.
- Messages are implementation and platform independent; their structure is defined in an abstract way to ensure that they can be composed with a program written in any programming language and running on any hardware and operating system. While an API can have similar properties, as OPHELIA showed with its CORBA interfaces, using messages, especially text-based ones, is easier to implement, to test and to deploy.
- Partial implementation of an API is not possible – trying to attach a partial implementation to an API-based system would result in errors sooner or later. Messages, on the other hand, can be sent or not, or can be processed or ignored, without any other participant noticing. That may not be desirable in many cases, but offers tool developers an opportunity to reap early benefits by implementing message communication in the most pressing parts of their tool landscape.

Message-based tool integration was also practised by [35], although the context was different: integrating a source code editor, debugger, configuration management tool and others into in IDE. The experiences presented in [36] teach that integrating tools directly, without any abstraction layer between them, is rather problematic. In [37], the authors suggest broadcasting change events to track artefact changes, which is exactly what artefact messages are designed for.

Essential to the messaging approach is a very limited set of different message types that consist of shared structures in order to allow for rapid implementation of a messaging module with minimum effort in order to access core functionality. Additionally, a defined and structured way of incorporating extra information into those messages – like “plug-ins” or “add-ons” for software – will facilitate the emergence of additional standards from usage, by contributions from the community maintaining the tools that integrate via these messages. To further these goals, the basic message format is XML, for which several mature parser implementations exist that can be leveraged to access the messages' contents without having to implement a text parser; extensibility is

also straight-forward, since one can always define optional elements.

The nature of messages is that they are transient – a tool sends a message and forgets about it, another tool receives it, perhaps interprets it and then discards it. A serious problem that comes with transience is that tools not attached to the message bus at the time the message was sent will never receive it. Some of the possible message buses mentioned above already counter that – a network file share can hold messages practically indefinitely – but others do not necessarily provide persistence inherently. Since not every participant interested in receiving the messages will be attached to the message bus at all times, the choice of medium depends on its ability not only to deliver messages sent while the participant was off-line, but also every message sent since the start of the project to any participant joining in at a later point.

5.1.2 Process Elements

The most important type of process element is definitely 'artefact'. As outlined in the introductory sections, tracing artefacts to their input artefacts ('product traceability') already provides serious benefits in regard to consistency and completeness of the project's product.

In order to trace as much information as possible about a single artefact without losing generality, artefacts are categorized into a few different generic types:

- *Documents* are human-readable text artefacts, possibly created from a template and therefore structured, but not (primarily) made for automated processing, but for consumption by project team members or stakeholders. Documents can have any machine format or exist only on paper, so long as their creation and each editing is messaged to the system. An example for a document is the Vision.
- *Reports* are automatically generated documents. Their contents are derived from other artefacts and formatted for use by human readers, in order to make certain data more easily available to the readers than in the original form. For example, a requirements document could be generated from the contents of a requirements database, a personal iteration plan for each team member could be generated from the planning tool's data etc. Since reports constitute snapshots of a specific version of the source data, they need not be version-controlled themselves. *Report templates*, which define the report format and which data is used on a report, may be treated as documents.
- *Models* are abstract representations of the system from a particular point of view, e.g. use case view, logical view (design model), implementation view or deployment view. Models consist of *model elements* that can be

arranged in *diagrams*.

- *Model elements* are abstract entities that represent a particular part of the system in a model, for example a class in the design model, or relationships between such entities. Model elements can be part of other model elements (such as a method is part of a class), or part of a model.
- *Diagrams* are visual representations of a model, part or whole. A diagram may change without changing the model, by rearranging the visible model elements or even adding or removing elements or their relationships from the diagram.
- *Records* are strongly structured pieces of data. While the term 'record' is often related to databases, records do not have to be stored in a database, but like database records, they have a fixed number of fields whose contents are restricted in type and possibly in the allowed values. Examples for records are functional requirements, change requests, bug reports and test cases.
- *Source files* are the primary products of a software development project – they contain the commands that constitute the software in (more or less) human-readable form. Test code and scaffolding are also written as source files, although they are not compiled as part of the final product. The main difference to documents is that source files are written not only for human readers, but also for automated processing (compilation or interpretation).
- *Binary files* are compiled source files. Like reports, they are automatically generated from other artefacts, so they need not be version-controlled independently. All the projects' binary files together (without test executables) form the executable software. Not every project will have binaries since scripting languages have become competitive, but they are prevalent enough to warrant their own type.
- *Configuration files* contain settings used to perform actions such as compiling (build configuration) or to access resources (e.g. database connection).

These generic types are supplemented by freely named sub-types, or specific types, that more precisely define the kind of artefact at hand. For example, “Vision” is a sub-type of “Document”, “C# source file” and “SQL Stored Procedure Script” are sub-types of “Source file”.

While artefacts and product traceability are essential, workers and their activities, which produce these artefacts, should not be forgotten. But since the proposal detailed in this thesis only encompasses the “first step” of artefact

traceability, they are only briefly discussed here.

A *worker* is a person relevant to the software development project – a team member or a stakeholder giving input or deciding issues. While in the RUP process model, workers take on one or more *roles* within the project, roles are omitted here because the sole focus is on the instance level. Workers would most likely be introduced into a project's traceability model by the project management tool, but each user-aware tool could theoretically notify the other participants of the user creating or manipulating an artefact.

Workers perform *activities* to produce artefacts; activities are instances of concrete pieces of work, most likely only relevant to project management – at least, until an activity is assigned to a worker, which could result in a corresponding notification to that worker. Finishing an activity is also a noteworthy event, since this enables project monitoring.

Unlike most artefacts, workers and activities are not version-controlled, but may be attributed a state – workers can be 'active' for the project, which means that they have time dedicated for project work, or 'inactive' – not yet on the project or (perhaps temporarily) not doing any project work any more. Activities, on the other hand, can be planned, running, or finished (cancelled activities can be deleted). To establish full process traceability, all state changes must be recorded and archived, which is another reason why the message bus should support message persistence.

5.1.3 Traceability Links

Artefact traceability means that between every two artefacts that are directly related somehow, there exists a record of that relationship so that information about it may be retrieved, i.e. there is a link from one artefact to the other that can be followed. These links can be modelled as directed arcs between nodes representing artefacts, and can therefore be expressed as a triple `<S, T, R>` where `S` is the source artefact, `T` is the target artefact and `R` is the relationship type. `S` takes the role `A` in the relationship, while `T` takes role `B` – the semantics of these roles are defined by `R`.

The relationship type consists – like artefact types – of a generic type and a specific type, where the generic type defines the behaviour and is taken from a limited set of available types, while the specific type is a freely named sub-type of the generic type used to clarify the semantics of the relationship. For a list of generic relationship types, UML [16] can serve as comprehensive source, where relationship types applicable to traceability can be taken from:

- Dependency: this is the most prevalent of relationship types in traceability, meaning that one artefact's contents depend on another

artefact's contents. This also means that if the dependee (role B) changes, the dependent (role A) may have to adapt. Until it is decided that no changes are necessary or the changes have been done, the relationship is said to be suspect, meaning that it is not certain that the current version of the dependent actually corresponds to the current version of the dependee.

- Aggregation: this denotes a part-whole-relationship, where role A means “part” and role B means “whole”. Unlike dependencies, these relationships do not become suspect, but may have to be deleted altogether if the part is removed from the parent. This may occur, for example, when a requirement is dropped from the release goals. The requirement can exist on its own, but is no longer a part of the requirements document for the current release; it may be moved to a to-do list and become a part of that, but since this is another target artefact, a new link must be established.
- Realization: the client (role A) is an implementation of the definition outlined by the supplier (role B). Examples include the source code for a class implementing a model element defining that class or a class model of a subsystem “implementing” a set of requirements, so this relationship type represents traceability along the specification axis, since the client is more concrete than the supplier. Relationships of this type behave like dependencies, but the advancement towards the final product realization links denote warrants an own type. Note, however, that a test script may be seen as realization of a test case, even if this relationship is not part of the path between requirements and final product.
- Association: This is the catch-all type for any link not covered by the others. Contrary to UML, associations are always directed (as opposed to symmetric) so that the roles are clearly defined, but this does not influence navigability. Examples of associations are: change request associated with the requirement to change, report associated with source data artefacts, test result associated with test case and binary file. Note that these examples always have a non-version-controlled artefact as source and a version-controlled artefact as target – the target is really a specific version of that artefact.

These generic types are subject to specialisation through specific types that hint at the semantics of a relationship between two specific artefact types. For example, a dependency between a test case and a use case will probably mean that the test case confirms the behaviour outlined in the use case, while a dependency between test case and binary file describes that the dependee is the

software to test. While the specific meaning of a traceability link will almost always be implicitly defined by the source and target artefact types, perhaps in combination with the generic relationship type, making the meaning explicit with an expressive relationship type name will help the people working with it to quickly attribute the correct meaning to any traceability link.

For different approaches to traceability link categorization, see [38], [39], [40] and [41].

5.2 Artefact Messages

Messages can signal the creation, modification or deletion of an artefact, where modifications essentially mean the creation of a new version. Apart from the kind of action performed on the artefact, every artefact message contains information to identify the artefact – a unique ID, revision number and the generic and specific artefact type. That is all the information necessary for basic traceability functionality in this kind of message. Extensions may define sets of artefact attributes to allow additional information to be shared between tools if desired, but each tool knowing which artefacts exist in a project is enough to enable the creation of traceability links.

```
<ArtefactMessage action="Create">
  <Sender clientId="MyCompany.RequirementsManager"
          userId="JohnDoe@example.com" />
  <Timestamp>2008-06-28T09:19:15.6410544+01:00</Timestamp>
  <Artefact id="MyCompany.RequirementsManager.UC001" revision="1">
    <Type generic="Record">Use-Case</Type>
    <Name>Browse Calendar</Name>
  </Artefact>
</ArtefactMessage>
```

Listing 5.1: Artefact Message Example

In addition, meta information about the sender and the time the message has been sent adds traceability to the action represented by the message. The following XML fragment shows a message about the creation of a use case with the name “Browse Calendar”:

The Sender element identifies the tool sending the message (ClientID) and the user performing the action (UserID). For ClientID, the naming convention is a dot-separated concatenation of the name or abbreviation of the company or organisation that created or maintains the tool, and the name of the tool or an abbreviation thereof – this should ensure unique tool identifiers, at least within any given project. The UserID attribute can be populated with any information the tool has that uniquely identifies its current user – a personal email address would be the most universal identification, but in some environments, it may be more desirable to use the current user's login name for the application or the

operating system.

The Timestamp element contains the date and time when the message was sent, in the locale of the machine from which it was sent. The date/time format is that endorsed in the World Wide Web Consortium's XML Schema Datatypes recommendation [42], which is based on the ISO 8601 standard.

The artefact itself is identified by a multi-part identifier consisting of the creating tool's ClientID and an artefact ID assigned by that tool. The tool is responsible for assigning unique identifiers to all its artefacts so that each artefact can be uniquely identified by the composed tool-artefact ID. The revision number is an integer that starts out with 1 at the artefact's creation and is incremented in each successive “modify” message. This enables other tools to refer to specific revisions of the artefact in their traceability link messages. Some artefact types may not be version-controlled, as mentioned before: change requests, test results and reports in general derive from or refer to specific versions of other artefacts, but do not have a revision number themselves. In this case, the “Revision” attribute value is “0”.

The artefact type is a specialisation of a generic type, in this case a record because a use case is essentially a data structure with attributes like name, actor and description. Note that type declarations are implicit, so there is no need for pre-agreed artefact types. Each artefact also has a human-friendly name in addition to its ID that other tools can use to present the artefact to their users. Artefact names do not have to be unique, but should be concise and descriptive enough for users to associate the underlying artefact with them.

5.3 Traceability Messages

Traceability links can be created and deleted, some of them – those that can become suspect – may be validated to remove the “suspect” status. A traceability link message contains the same meta-information as an artefact message, the payload data consists of the source artefact, target artefact and the relationship type.

The XML fragment in Listing 5.2 shows a message reporting the creation of a traceability link between the test case “Navigate to next month” and the use case “Browse Calendar”. Compare the representation of the traceability link to that used in [43].

The root element is named “ArtefactArtefactRelationshipMessage” with future expansions in mind that may declare other traceability links between artefacts, workers and activities. In this first step to achieve artefact traceability, the only links established are between two artefacts, hence the name.

```
<ArtefactArtefactRelationshipMessage action="Create">
  <Sender clientId="OtherCompany.TestTracker"
          userId="JohnDoe@example.com" />
  <Timestamp>2008-06-29T15:42:36.2461799+01:00</Timestamp>
  <ArtefactArtefactRelationship>
    <Type generic="Dependency">confirms implementation</Type>
    <RoleA name="Test">
      <Artefact id="OtherCompany.TestTracker.TC001" />
    </RoleA>
    <RoleB name="Requirement">
      <Artefact id="MyCompany.RequirementsManager.UC001"
        revision="1" />
    </RoleB>
  </ArtefactArtefactRelationship>
</ArtefactArtefactRelationshipMessage>
```

Listing 5.2: Relationship Message Example

Traceability links do not have identifiers themselves, but derive their identity from their source and target artefacts as well as their type. There can be only one traceability link of a specific type between a source artefact and a target artefact. The revision number in the target artefact is relevant for determining the traceability status (“suspect” or not) or the target revision in associations. When validating a suspect relationship, a message is sent using the last known revision number of the role B artefact. Since the role A artefact's revision number is irrelevant for suspect status, it is entirely optional, and therefore omitted here.

Like artefact types, relationship types are declared implicitly within traceability link messages – there is no central authority or separate negotiation of specific relationship types.

5.4 Alternative Message Formats – Semantic Web Languages

The XML-based message formats presented in the previous chapters are concise and extensible, but use newly defined XML schemas. Semantic Web Languages have been invented to describe relationships between web resources and anything else that can be identified with a URL, and assign meaning to these relationships. This chapter will explore how the World Wide Web Consortium's Resource Description Framework (RDF) could be used to express the same basic information as the message formats presented above. While at least one other semantic web language exists (Basic Semantic Web Language, [44]), this particular language is a less suitable format than RDF, and competing web languages in general have no relevance next to the RDF's XML schema.

The Resource Description Framework is a language to formulate assertions about resources and their relationships. It is based on a graph data model, where each assertion includes two nodes and an arc. The two nodes represent

subject and object in the assertion, while the arc represents the predicate. Both nodes and arcs are generally identified by a URI, although nodes can have an arbitrary literal (such as a person's name) as identifier or even be “blank”, meaning that the node has no identifier at all. Blank nodes can serve as conjunction operator for a set of relationships, for example in an “editor” relationship between a web site and a person that has a name and a home page, the “editor” relationship would point to a blank node, which would have “fullName” and “homePage” relationships to the respective nodes. The XML representation of that small tree, after applying some syntax short-cuts, looks like Listing 5.3.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
   <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                    dc:title="RDF/XML Syntax Specification (Revised)">
      <ex:editor>
         <rdf:Description>
            <ex:homePage rdf:resource="http://purl.org/net/dajobe/" />
            <ex:fullName>Dave Beckett</ex:fullName>
         </rdf:Description>
      </ex:editor>
   </rdf:Description>
</rdf:RDF>
```

Listing 5.3: RDF Example

In English: the web site “http://www.w3.org./TR/redf-syntax-grammar” with the title “RDF/XML Syntax Specification (Revised)” has an editor whose name and home page are “Dave Beckett” and “http://purl.org/net/dajobe/”, respectively. The RDF element serves as a container that also defines the name-spaces used in the assertions. (The example has been taken from [BEC04].)

Using these simple ingredients, complex relationship trees or forests can be established. Fortunately, traceability messages only describe artefacts or a single relationship between two artefacts, and thus require only small structures. suggests an RDF-compliant version of the “create artefact” message shown in Listing 5.1. Note that the message itself has no identifier, wherefore the top node is a blank one – Figure 5.1 shows the corresponding graph.

The conceptual problem with this approach is that assertions about artefacts may contradict each other, since the artefact's name, for instance, may change. While it is clear that the assertion of the later message should be viewed as valid in case of doubt, keeping the old assertion posted (so that the history of the artefact is available) causes a false fact to remain asserted. While this is not really relevant to the practical implementation, it is a reason – at least for purists – to refrain from employing RDF as messaging format.

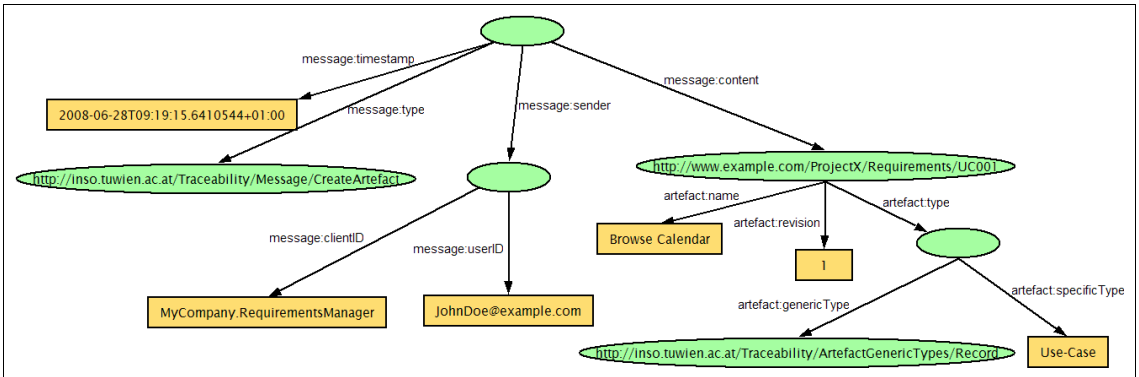

Figure 5.1: Create Artefact Message, RDF Graph

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:artefact="http://inso.tuwien.ac.at/Traceability/Artefact/"
         xmlns:message="http://inso.tuwien.ac.at/Traceability/Message/">

  <rdf:Description message:timestamp="2008-06-28T09:19:15.6410544+01:00">
    <message:type
       rdf:resource="http://inso.tuwien.ac.at/Traceability/Message/CreateArtefact" />
    <message:sender>
      <rdf:Description message:clientID="MyCompany.RequirementsManager"
                       message:userID="JohnDoe@example.com" />
    </message:sender>
    <message:content>
      <rdf:Description rdf:about="http://www.example.com/ProjectX/Requirements/UC001"
                       artefact:Name="Browse Calendar"
                       artefact:Revision="1">
        <artefact:type>
          <rdf:Description>
            <artefact:genericType rdf:resource=
                "http://inso.tuwien.ac.at/Traceability/ArtefactGenericTypes/Record" />
            <artefact:specificType>Use-case</artefact:specificType>
          </rdf:Description>
        </artefact:type>
      </rdf:Description>
    </message:content>
  </rdf:Description>
</rdf:RDF>
```

Listing 5.4: Artefact Message Example in RDF

Apart from that, there are subtle differences in the semantics of the messages: while the original message format restricts generic artefact types by using an enumeration data type, the RDF format presented here references a URI instead, making it more extensible, but on the other hand less robust if a new generic type of artefact needed special treatment in any application. Note, however, that since RDF allows the specification of an arbitrary data type for each literal, an alternative format could be given to reach the same closure of the value space for generic artefact types.

Another difference is that the RDF message type “Create Artefact” includes both

the information that this is an artefact message and that the action is “create”. This has been done to keep the RDF message concise and robust. Splitting the action from the message:type attribute would create a message:action attribute, which would point to a URI representing a sub-entry of the message:type URI, another constraint that would have to be enforced. For two message types and three actions each, this would mean over-engineering the message format.

Relationship messages in RDF can of course be done, but are omitted for brevity.

6 Implementation Prototype

In order to demonstrate the viability of the message-based traceability approach in a scenario relevant to everyday software development, prototype tools have been implemented. These tools are a requirements management tool and a test tracking tool that communicate through the outlined message formats and present traceability information to their users. Requirements and test tracking have been chosen because historically, these disciplines have been the first to be brought together by traceability, so the conclusion at hand is that this is a most pressing application.

The tools have been built to support a minimal workable set of features, then had modules added to enable communication, processing of messages and persistence for received information. The required change to the existing application has been minimal, as they consisted mainly of adding code to trigger message sending and additions to the user interface to present traceability information. This has been done to show that existing applications can be extended easily, with reusable components and a few additions to the GUI.

As target platform for the prototypes, the .NET framework was chosen (starting with 2.0 and upgrading to 3.5 during the project), with C# as programming language. This had little to do with actual advantages of the platform in regard to the field of application, but rather with personal preference. Apart from that, the open source communities of the Java platform and the C++/Linux platform are vivid enough to reproduce and improve the code necessary for connecting an existing tool to a message bus, while the .NET community is rather small and still in the stage of formation. This is also the reason why new prototype tools have been developed, rather than modifying existing ones – there is a distinct lack of open source software engineering tools in the .NET world, at least if sourceforge.net is accepted as a good indicator.

XMLBlaster is used as the message bus. This is an open-source message-oriented middleware that consists of a server that relays messages of different topics to subscribed clients of the message's topic. A more detailed discussion of XMLBlaster, its incorporation into the prototype set-up and the possibilities and problems associated with that software is part of the following sections.

The following chapters will give an overview of the RequirementsManager and the TestTracker tools, outlining their self-contained functionality and their software design. After a presentation of XMLBlaster, discussion will focus on how a reusable messaging component using XMLBlaster as message bus has been built, and how the prototype tools were extended to send and receive messages. Finally, the additions to the tools' GUI that allow the user to access and enter traceability information are presented.

6.1 Requirements Manager

This tool was built to allow the management of use cases and associated actors, as well as features and constraints that may or may not derive from these use cases. Figure 6.1 shows the general layout by example of the use case view: on the right, there is a master view controlling the contents displayed on the left, where the attributes of the selected item can also be edited and saved, or the changes since the last save can be discarded by clicking the corresponding button on the bottom right.

The following sections will detail the tool's features and architecture.

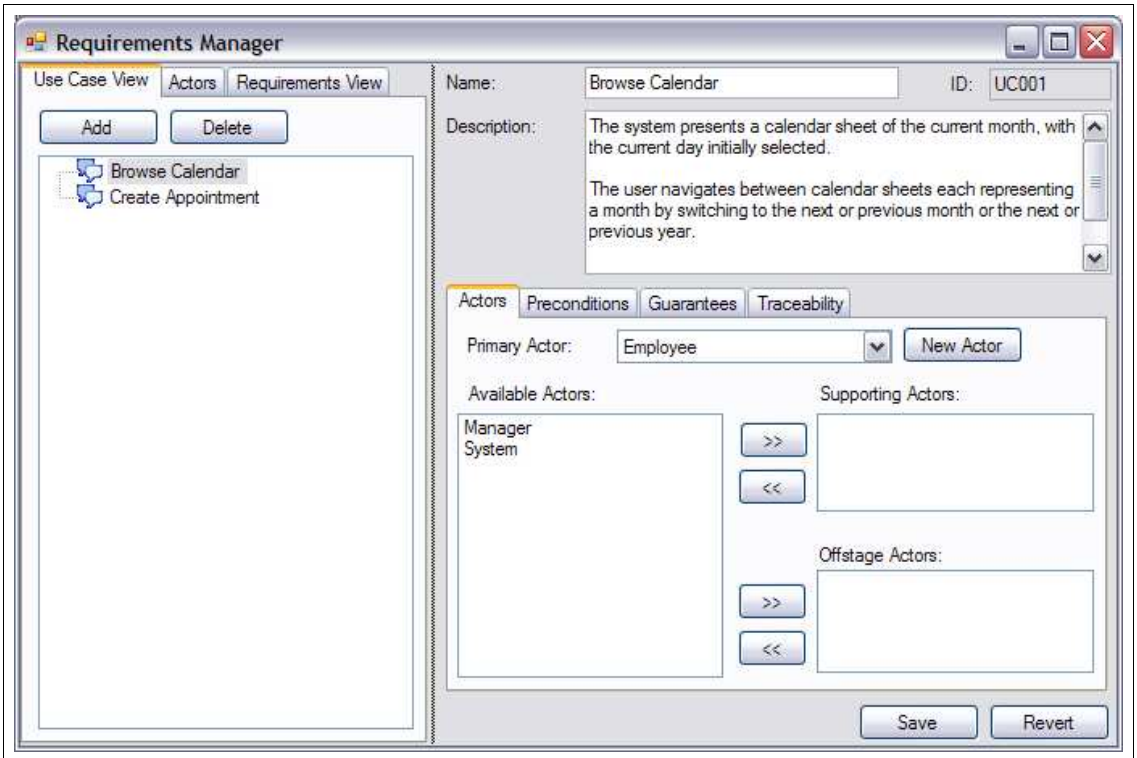

Figure 6.1: Use case view

6.1.1 Functionality

The Requirements Manager can be used to create, edit and delete use cases, features and constraints – these three artefacts are also those chosen for the traceability demonstration. The fourth artefact – actor – is only used internally; use cases can have actors assigned to them, the primary actor being the one starting the use case, while the supporting actors play a role in use case execution and off-stage actors do not actively participate in the use case, but have vested interests in how the use case is executed.

All three primary artefact types share some attributes like an identifier, a name

and a description. The identifier is actually an integer; what is seen in Figure 6.1 in the ID box is the "public" identifier used in the traceability demonstration, which includes type information – in this case, "UC" stands for "use case" - so that artefacts with the same internal id (since each type has its own uniqueness constraint, so there can be one use case and one feature each bearing the id 24, for example) can be disambiguated. Name and description are free-form strings.

Use Cases

In addition to these standard attributes, use cases can have preconditions, meaning conditions that have to be met in order for a particular use case to be started. This is implemented as a list of strings, where each string is a precondition. See Figure 6.2 for an example.

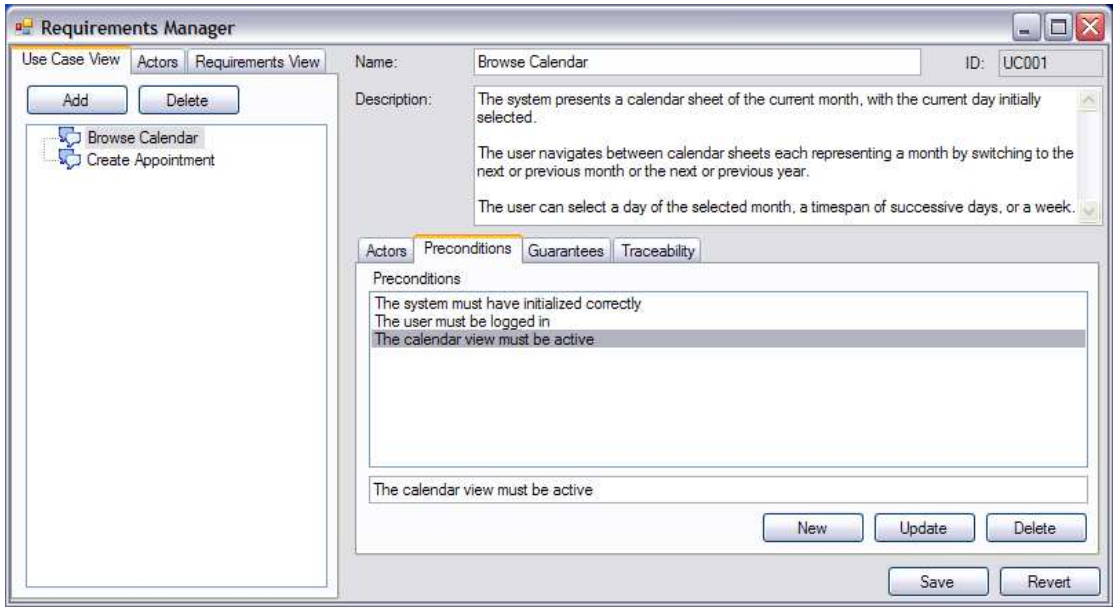

Figure 6.2: Use case preconditions

Use cases can also guarantee some results, where minimal and success guarantees are distinguished. Minimal guarantees are met even if a use case fails – these guarantees are mostly concerned with maintaining data consistency and system integrity. Success guarantees are only certain to be met if the use case has been executed successfully. Guarantees are implemented similarly to preconditions, as depicted in Figure 6.3.

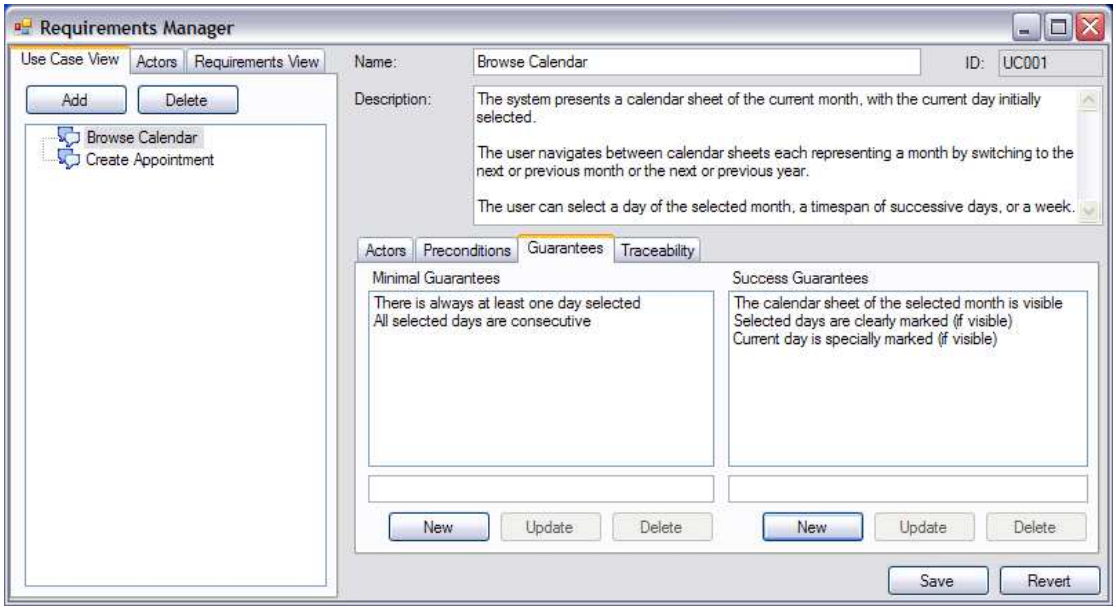

Figure 6.3: Use case guarantees

Actors

The actor management can be accessed by activating the “actors” tab in the master view section of the window. Actors are listed alphabetically and can be

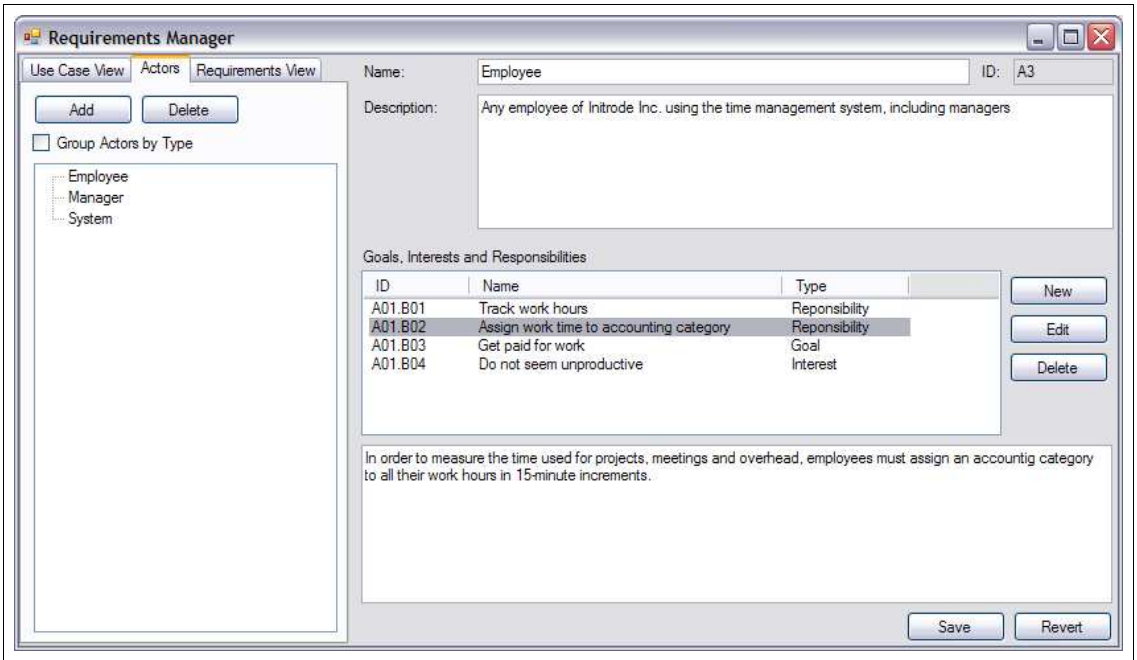

Figure 6.4: Actor view

grouped by actor type (person, organization, machine). They share the basic attributes of use cases – id, name and description – but an actor also has behaviours, which are classified as goals, interests and responsibilities. These behaviours also have an id, name and description each, plus the type of behaviour.

An actor's behaviours are shown in the list view in Figure 6.4, except for the behaviour description, which is shown in the text box below for the selected behaviour. This is necessary to deal with lengthy descriptions that would not fit into a list view column.

Requirements

The final master view is titled "requirements" and encompasses features and constraints, the former meaning functionality that the system has to implement, while the latter put boundaries on certain aspects of the design or implementation.

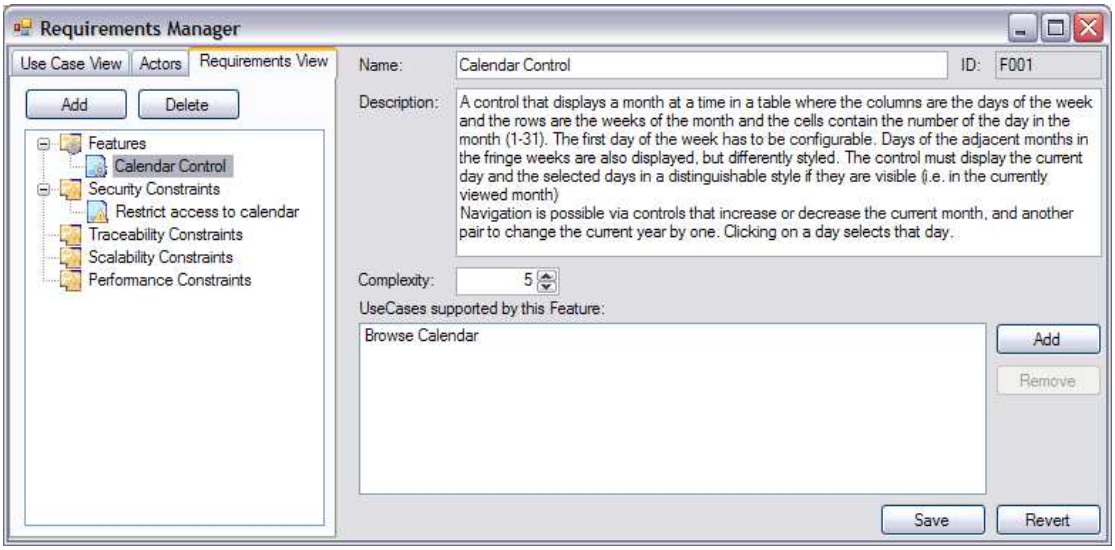

Figure 6.5: Feature view

Features can be organized hierarchically, so that a complex feature may be split into several more manageable features. Constraints are grouped into categories: security, traceability, scalability and performance. Both features and constraints can be linked to use cases, so that it can be traced which use cases a feature supports and which use cases are affected by a certain constraint. Features also have an abstract complexity attribute that can be used to enter estimates in any metric. Figure 6.5 shows a feature details view, while Figure 6.6 shows a

constraint details view.

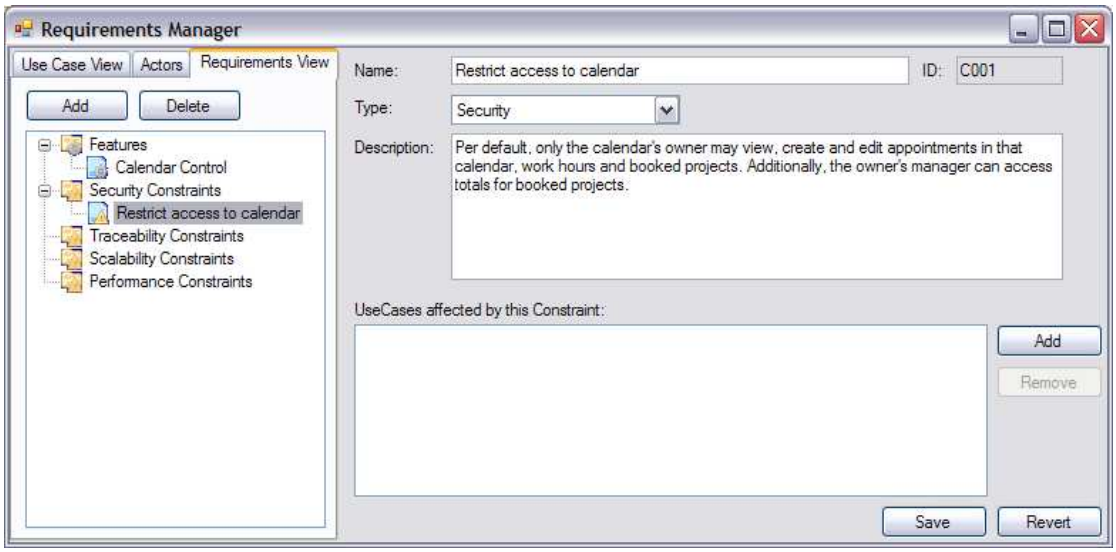

Figure 6.6: Constraint view

6.1.2 Software Architecture and Design

Both Requirements Manager and Test Tracker have been constructed using a three-layer architecture of persistence, business logic and presentation layers.

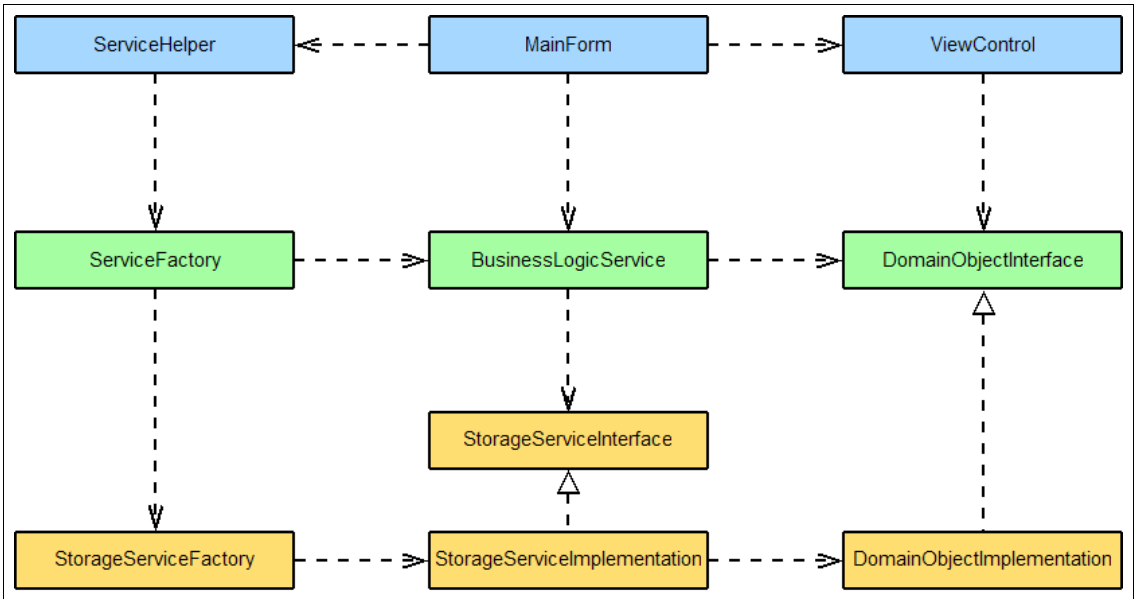

Figure 6.7: Software architecture

Data is stored in Microsoft SQL Server Express databases, retrieved and written by stored procedures which are called from the persistence layer. As a rule of thumb, there is one storage service class per strong entity that handles object-relational mapping between the application and the strong entity as well as any associated weak entities. The objects constructed from the data records are handed to the business logic layer as single objects or lists of objects of the same class, so the responsibility of putting together views of related objects for the persistence layer falls to the service classes of the business logic layer. The presentation layer can perform simple manipulations such as setting property values by itself, but more complicated or structural changes to the model are performed by the business logic. Figure 6.7 shows a diagram of the overall software architecture of the Requirements Manager and Test Tracker applications.

Persistence Layer

The design goal was to provide a complete abstraction of the underlying persistent storage type. In order to achieve that, only interfaces are exposed to the client, while the factory class can choose the correct implementation to return based on the connection string settings supplied by the client. The connection string settings are stored in the application configuration of the Requirements Manager and include an attribute that determines the database provider. The factory class is initialized with the corresponding ConnectionStringSettings object to ensure that all storage service classes created by the factory share the same configuration. (The factory pattern of software design is described in [45])

At this time, there is only an implementation for the SqlServer provider, but support for other database systems can be easily added – all it takes is a set of classes implementing all the storage service interfaces and slight modifications to the factory class to make it aware of the implementation and map the corresponding provider name to that implementation; the upper layers are not concerned at all.

Powerful and simple as this design is, it has its drawbacks. If a large number of storage service classes are needed, the effort required to write all the corresponding methods of the factory class will become too great to consider this approach practical, especially if there are several storage providers to support – adding an implementation for an additional provider would cause significant cost for the factory class alone. A code generator may help, but an even more elegant solution would certainly be instantiating storage service classes via reflection. This would only require mapping a provider name to a sub-namespace of the factory class's namespace and getter method

implementations like this:

```
private enum StorageService { UseCase, Actor, Behaviour, Feature, Contraint }
public IUseCaseStorageService GetUseCaseStorageService()
{ return (IUseCaseStorageService)GetImplementation(StorageService.UseCase); }
```

The GetImplementation method would determine the sub-namespace from the provider name given in the ConnectionStringSettings object and the class name from the given StorageService enumerator value. The class would then be instantiated and initialized with the actual connection string. As further generalization, instead of providing one method per storage service, a generic method could determine the class name from its type parameter, which specifies the expected return value interface:

```
public TStorageService GetStorageService<TStorageService>()
```

Since there are only five storage service classes necessary for the Requirements

```
public class StorageServiceFactory
    {
        public StorageServiceFactory(ConnectionStringSettings connectionString)
        {
            _connectionString = connectionString.ConnectionString;

            switch (connectionString.ProviderName.ToLowerInvariant())
            {
                case "sqlserver":
                    _provider = ProviderType.SqlServer;
                    break;
                default:
                    _provider = ProviderType.Unknown;
                    break;
            }
        }

        private readonly string _connectionString;
        public string ConnectionString
        {
            get { return _connectionString; }
        }

        private enum ProviderType { Unknown, SqlServer }
        private readonly ProviderType _provider;
        private ProviderType Provider
        {
            get{ return _provider; }
        }

        public IUseCaseStorageService GetUseCaseStorageService()
        {
            switch(Provider)
            {
                case ProviderType.SqlServer:
                    return new SqlServer.UseCaseStorageService(ConnectionString);
                default:
                    throw new NotImplementedException();
            }
        }
[...]
```

Listing 6.1: Storage service factory

Manager application, these designs have been rejected in favour of a simpler and more easily understandable implementation.

The storage service classes all derive from a provider-specific abstract base class that stores the connection string and provides the connection object and resource management methods, notably the Finalize method that frees connection, command and data reader resources. There are, however, no generic methods for retrieving or persisting data – the same argument as for the factory class design also applies here.

Domain Model

The objects of the domain model component are created from data records by the storage service classes or from scratch by the service classes of the business logic layer. They represent the artefacts that are managed by the application, in the case of the Requirements Manager use cases, actors, features and constraints. Furthermore, there may be subordinate objects like behaviour, which is subordinate to actor.

The classes contain information that is irrelevant to the application, but necessary for the business logic to assemble objects into views, i.e. reference identifiers. Furthermore, some properties are not intended to be set by the application – especially the identifier – but have to be set by the storage and business service classes. In order to restrict access to these properties and manage the visibility of low-level properties, domain model classes implement interfaces that contain only the information necessary for the presentation layer. The application front-end is not aware of the implementation, i.e. there is no direct dependency on the implementation library, but only on the interface library, which is a separate component.

This does open up the theoretical danger of the front-end passing objects that are not “native” domain model objects, but implement the interface anyway, to the business logic layer, resulting in failure and possibly application crashes. However, since the front-end and back-end have been developed together, this is not an issue in the current application. Alternative front-end implementations – for example a web application – would have to restrict themselves to only passing objects obtained from the back-end to the business logic in order to function properly; this is only an issue of documentation, since malevolently misbehaving applications cannot be considered a real problem.

Business Logic

Similar to the persistence layer, the business logic layer consists of a factory class responsible for instantiating the service classes. A significant difference is

that the business logic layer does not have to deal with multiple implementations of services, so there are no service interfaces and the service classes themselves are marked public. However, they cannot be directly instantiated since their constructors are internal, and thus not visible outside the component – the service factory has to be used by the application front-end.

```
public class ServiceFactory
    {
        public ServiceFactory(ConnectionStringSettings connectionString)
        {
            _connectionString = connectionString;
            _storageServiceFactory = new StorageServiceFactory(connectionString);
        }

        private readonly ConnectionStringSettings _connectionString;
        private ConnectionStringSettings ConnectionString
        {
            get { return _connectionString; }
        }

        private readonly StorageServiceFactory _storageServiceFactory;
        private StorageServiceFactory StorageServiceFactory
        {
            get { return _storageServiceFactory; }
        }

        public UseCaseService GetUseCaseService()
        {
            return new UseCaseService(StorageServiceFactory);
        }
    [...]
}
```

Listing 6.2: Business logic service factory

Like the storage service factory, the service factory is a class that takes a configuration parameter at the constructor, in this case the ConnectionStringSettings object needed to initialize a storage service factory. In a simple application like this, the service factory does not require any configuration for instantiating the business logic services, so it makes sense to take only the connection string settings as configuration parameter. In more complex and configurable systems, a custom configuration class – probably implemented in the same component as the service factory itself – would be a better alternative.

The service classes are initialized with the factory object they use to create the necessary storage services – the connection string itself is not relevant them. The ConnectionString property of the factory class is used for instantiation of the messaging service, which will be discussed later. Listing 6.2 shows that using the provided storage service factory class, the UseCaseService can initialize the storage service as well as the ActorService it depends on.

```
public class UseCaseService
{
        private readonly ActorService _actorService;
        private readonly IUseCaseStorageService _storageService;

        internal UseCaseService(StorageServiceFactory storageServiceFactory)
        {
            _storageService = storageServiceFactory.GetUseCaseStorageService();
            _actorService = new ActorService(storageServiceFactory);
        }

        internal IUseCaseStorageService StorageService
        {
            get { return _storageService; }
        }

        internal ActorService ActorService
        {
            get { return _actorService; }
        }
        [...]
}
```

Listing 6.3: Service class example

Presentation

The actual executable assembly is mainly concerned with displaying available information and providing means to edit artefacts to the user. Requirements Manager consists of a main form that consists essentially of a tab control and an empty panel. The tab control contains one master view per tab, while the empty panel is used for the detail view controlled by the master view on the active tab. For example, if the use case view is active and a use case node has been selected, the UseCaseMasterView instantiates and initializes a UseCaseDetailView object with the appropriate UseCase object and signals the main form that this new detail view should be displayed. The main form then removes the current detail view from the panel (if there is one) and attaches the new detail view to it.

Both master and detail views are implemented as separate classes derived from UserControl. Since the use case detail view is rather complex, parts of it have been moved to their own classes as well, most notably the contents of each tab in the lower half of the view. This is also the case for the behaviour area of the actor detail view. For the requirements master view, there are separate detail views for features and constraints.

Dialogues are used to create the artefacts; they let the user enter the data necessary for creation of the artefact and validate the user's input. Each dialogue is a separate class derived from a common Dialogue base class containing the OK, Apply and Cancel buttons, which derives from System.Windows.Forms.Form.

6.1.3 Database

The database structure is rather simple since there are only four strong and one

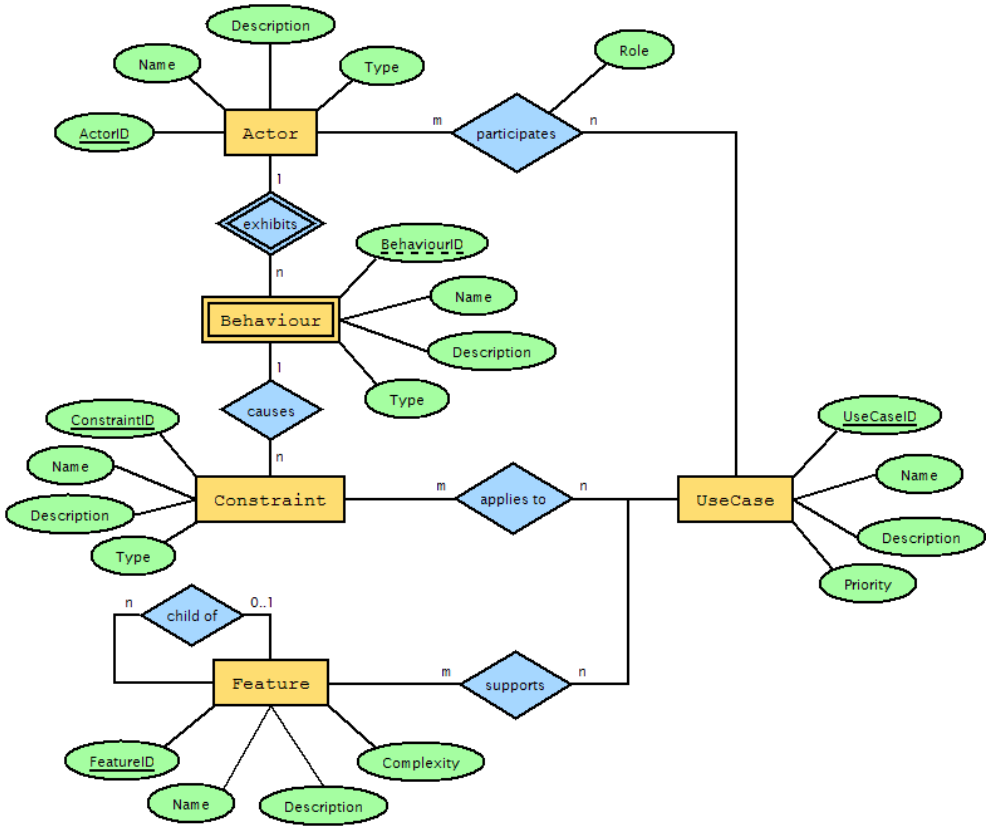

Figure 6.8: ER diagram for Requirements Manager database

weak entity with only a few fields each:

The relationship between use case and actor contains the role attribute, which determines if the actor is primary, supporting or off-stage for the use case. While the database structure would allow more than one primary actor for one use case, that is not desired and has to be prevented by the storage and business services.

A behaviour is always associated with an actor and cannot exist without it, which is why the actor identifier is also part of each behaviour's key. Features can be arranged in a tree, which is why each feature may have a parent feature. The rest of the database design is straightforward; cardinalities of m or n always mean zero or more.

6.2 Test Tracker

This tool serves two purposes: first, defining test cases and arranging them in test suites and second, track test runs and record test results. The main window has a layout similar to Requirements Manager and is used for the first task:

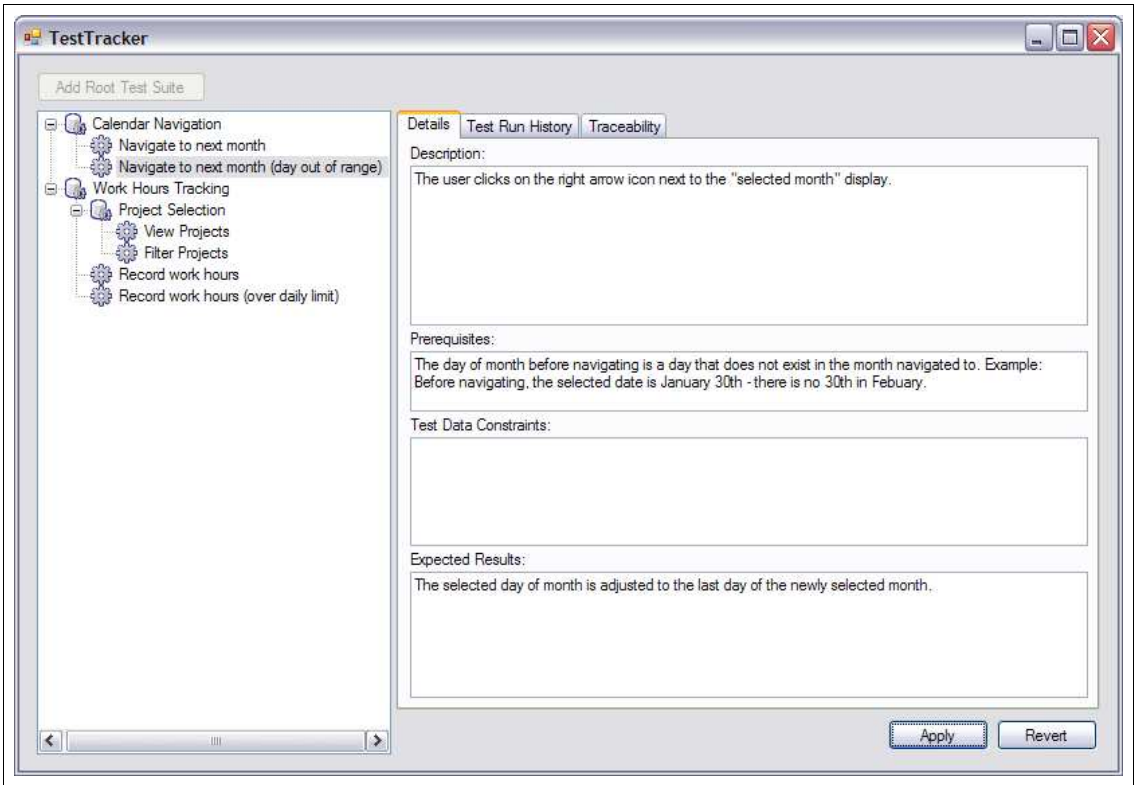

Figure 6.9: Test management in Test Tracker

Like in Requirements Manager, the master/detail pattern is applied, with the detail view also containing the command buttons to save the current window contents or revert to the last saved version.

The second task, recording test runs, is done in a separate modal window (see Figure 6.10) that functions likewise, but offers different command buttons since a test run is created once and cannot be edited later.

6.2.1 Functionality

Test suites can be nested to any desired level in order to group test cases into sets that will be executed in a test run. Test cases can, however, only belong to one test suite. While a test suite only features a general description on the details tab, a test case contains additional information for the tester:

prerequisites for the test case to be executable, constraints on the test data to use for the test, and the conditions for a successful test – the expected results.

A test run history is available for both test cases and test suites. A test run is a collection of test case executions; in case of a test suite, all directly or indirectly contained test cases are run, but a test case can also be run individually. Recording a test run for a suite is shown in Figure 6.10 – as can be seen, a test case is assigned one of the results success, failure or skipped, the latter meaning that the test case has not been executed. In the tree on the left, the current overall status of the test suite and the results of the already executed test cases are shown by icons – in this case, success – while the open test cases retain their standard icon. Clicking on a result button (pass, fail, skip) will set the result and thereby the icon in the tree view, and automatically select the next open test case.

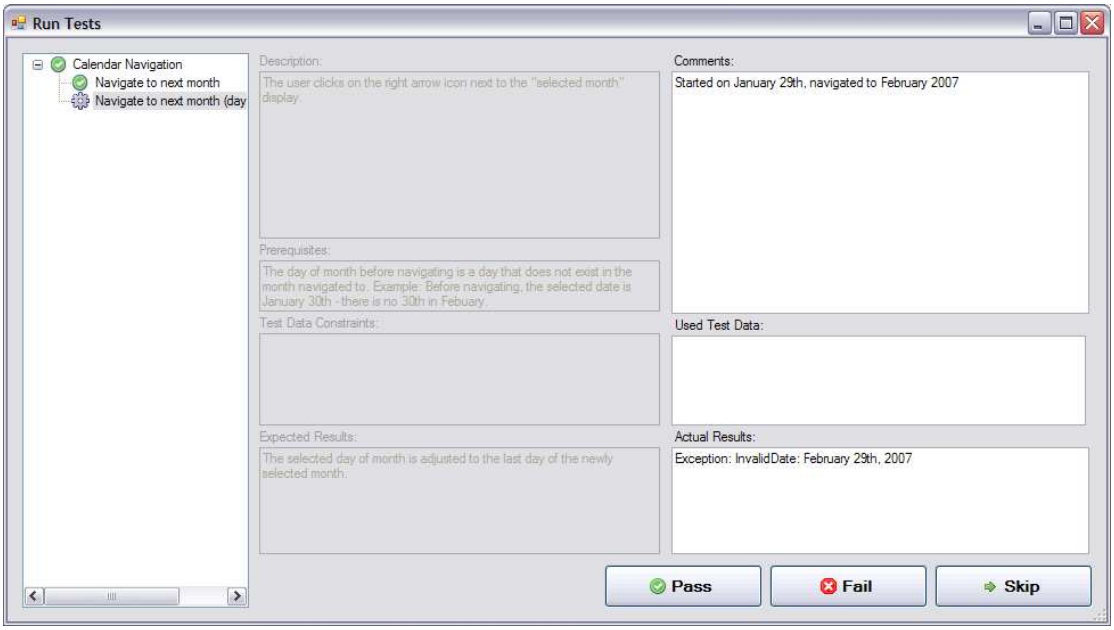

Figure 6.10: Recording a test run

A test run can be wholly aborted by closing the window before all test cases have been assigned a result state. After the last test case is assigned a result, the test run is saved and the window closed after a message to the user.

The test case details in the left part of the detail view are displayed for the currently selected test case. Comments, used test data and actual results can be recorded specifically for that test case.

Once test runs have been recorded, they can be viewed in the test run history tab of the test suite and test case detail views – see Figure 6.11 for an example. The overall result of a test suite is determined by the "worst" outcome of all its

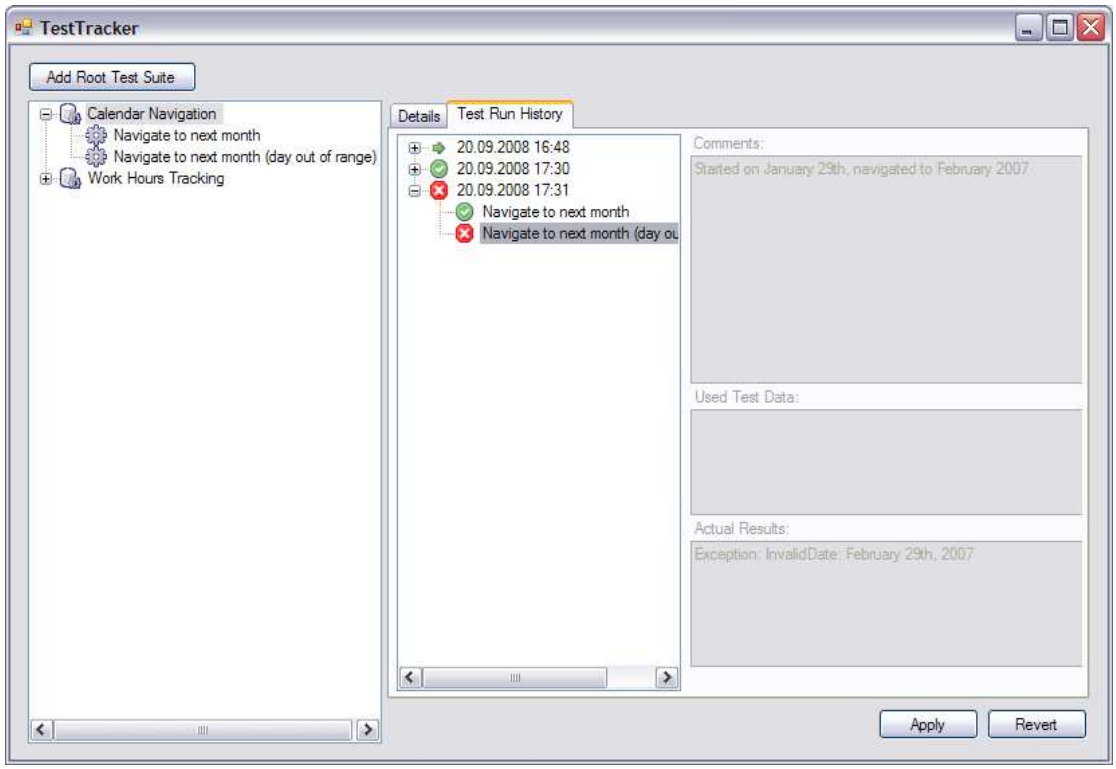

Figure 6.11: Test run history

children – failure, skipped and success in order of precedence. This means that in a tree of test suites where one test case fails, the root test suite is in status "failure", but subordinate test suites not containing the failed test case can be in "success" or "skipped" states. This scheme has been borrowed and adapted from NUnit, a unit testing framework for .NET (see http://www.nunit.org/).

6.2.2 Database

Since the software architecture of Test Tracker is practically identical to Requirements Manager, its discussion is omitted. Test Tracker's data model is even simpler than that of Requirements Manager, but is shown in Figure 6.12 for completeness.

The only non-obvious part is the combination of the two attributes TestType and TestID of the entity TestRun. Since a test run can refer to a test suite or a

test case, the reference must be stored this way – the TestType attribute determines whether the reference target is a test suite or test case, and the TestID attribute functions as the "foreign key" field to the table. Of course, there is no foreign key and therefore no referential integrity possible with this construct.

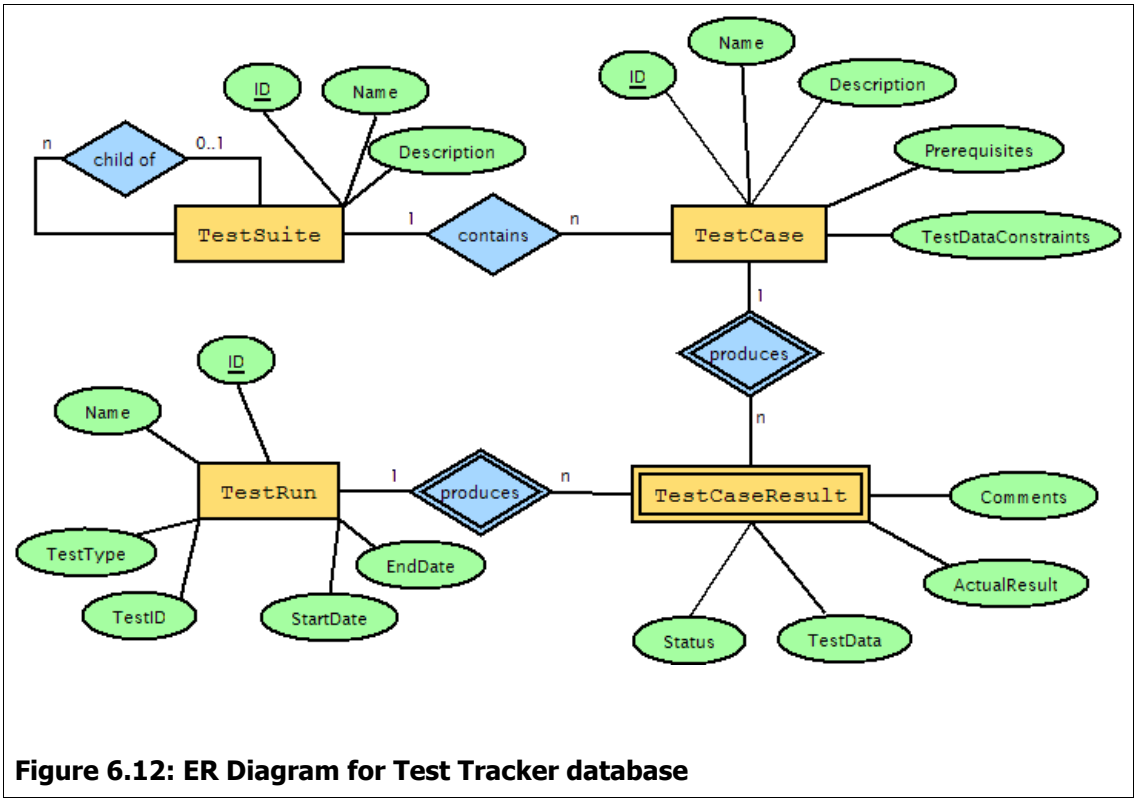

Figure 6.12: ER Diagram for Test Tracker database

Alternatively, an additional entity "test" could have been introduced in order to share the value space for the ID field among test suite and test case – these entities would then reference the "test" entity which would contain the identifier, name and description fields (since they are common to both these entities), while the existing entities would only retain the attributes not found in the other.

While this mimicking of inheritance in a relational database is possible and has its applications, the complexity associated with data storage and retrieval outweighs the benefits of referential integrity in this case. Therefore, the presented design variant has been chosen.

6.3 Messaging

Since both tools the prototypical implementation consists of share the same platform, it is a natural consequence to implement a messaging module for that

platform that both tools can use. This section presents the components used to create message objects, send the messages they represent via XML Blaster and receive these messages, converting them to message objects for the applications to interpret.

6.3.1 Client

This component handles sending and receiving of messages. A factory class constructs a messaging client object based on the configuration object it is given and connects the client object to an application-specific message receiver. The receiver has to implement an interface that consists of two methods: receiving an artefact message and receiving a relationship message. Both methods do not take raw XML or DOM objects as parameter, but rather message objects tailored to easy access to the relevant information contained in the messages.

At this time, there is only a client implementation to handle connections to XML Blaster, but the design allows for additional implementations dealing with different communication buses to be included and made available without major refactoring. The configuration object should be generic enough to allow a wide range of services to be used. See Listing 6.4 for the implementation of the factory class and the configuration class.

Each client implements an interface that allows the application to control

```
public static class ClientFactory
{
   public static IClient GetClient(MessagingConfiguration configuration,
                        IMessageReceiver receiver)
   {
      switch (configuration.ServerType)
         {
         case ServerType.XmlBlaster:
            return new XmlBlasterClient(configuration, receiver);
         default:
            throw new NotImplementedException("No implementation available
                              for server type " + configuration.ServerType);
         }
      }
   }
}

public enum ServerType{ Unknown, XmlBlaster }

public class MessagingConfiguration
{
   public ServerType ServerType { get; set; }
   public string ServerName { get; set; }
   public int ServerPort { get; set; }
   public string ClientName { get; set; }
   public int ClientCallbackPort { get; set; }
   public string UserName { get; set; }
   public string Password { get; set; }
}
```

Listing 6.4: Messaging client factory and configuration

connecting to and disconnecting from the server, actively requesting messages (in contrast to waiting for an event notification) and sending messages, as well as checking the connection status. Depending on the communication bus, message push may be available, meaning that whenever a message is placed on the bus, listening clients are passed the message via event handling. If this is not possible, either applications have to poll the messaging client for new messages, or the messaging client does the polling and pushes the received messages to the

```
public enum ConnectionStatusFlags
{
   disconnected = 0,
   connected = 1,
   listening = 2
}

public interface IClient
{
   bool IsConnected { get; }
   bool IsListening { get; }
   void FetchMessages();
   void Connect();
   void Disconnect();
   void SendMessage(IMessage message);
}
```

Listing 6.5: Client interface

application. The connection status tells the application if message pushing is active in addition to whether it is connected and thus can send and actively fetch messages. Listing 6.5 shows the interface a client must implement.

XML Blaster Client

XML Blaster is an XML-based message-oriented middleware. It consists of a Java server and several client implementations. Unfortunately, no .NET implementation is publicly available, but a managed C++ wrapper exists for the C client, importing its functions via [DllImport]. For the messaging prototype under discussion, a C# wrapper implementing the client interface above has been written. After several failed attempts to connect the wrapper to the original client by referencing the binaries compiled in the solution provided by xmlblaster.org, including the C and C++ projects of the XML Blaster community into the Visual Studio solution of the prototype proved to be workable.

In general, setting up XML Blaster was far from easy or intuitive. Efforts to run the Java-based server as a windows service failed as well as enabling message persistence in a Microsoft SQL Server database, although instructions are available and were followed. While the limitations of having to run the server in console mode and losing all posted messages with a shut-down are irritating but acceptable, the message distribution seemed to only work if both tools were

connected. Connecting with one tool after the other had posted a message would result in the late-connecting tool not receiving the message, even with an active fetch. While it may be assumed that XML Blaster provides means to circumvent this limitation, finding instructions is rather difficult, and executing troubleshooting instructions does not lead to success with acceptable certainty.

Server configuration is done strictly through text-based configuration files, not unlike the Apache web server. However, comments in the pre-made configuration file are not all that helpful, pointing to on-line documentation instead. As has been said before, even with that documentation, it was not possible to set up message persistence to a MS SQL Server database, although it is officially supported.

Adding to frustration with that product is the fact that programmatic configuration of the client is verbose, partly because defaults are not clear so many parameters have to be included although they might be superfluous.

6.3.2 Messages

This component is responsible for creating message objects from an artefact or relationship and serializing and de-serializing message objects. For the first task, a message factory is provided that offers two methods for creating an artefact message and a relationship message. Both methods require a sender object and an action as parameters in order to initialize the message objects. The third parameter necessary to construct a message is an object implementing an interfaces suitable for artefacts or artefact-artefact relationships. The interfaces are defined in the component and shown along with the factory class in Listing 6.6.

In contrast to the application-specific services, the message factory accepts any object that implements the appropriate interface. Internal objects are created from the provided objects in order to control serialization. This is useful for applications, since they do not have to create the content objects from a factory first, but can use their own objects if they implement the relevant interfaces.

Note that message objects are, however, always built by the client. Message objects can be seen as wrappers around the content objects (artefact and relationship) that also contain the message header (sender information, time stamp and the action performed on the contained object). The client builds message objects by assembling the necessary information from the given parameters, or the serializer builds them from a stream after the client received a message. There are no methods in the client that accept pre-made messages as parameters.

The object model is oriented towards serialization with Microsoft's

XMLSerializer class, which also explains the XML annotation attributes in the sender structure, which is used in each message header. In order to obtain the message formats presented in chapter 6, in addition to the content objects – artefacts and relationships – some helper classes are used, whose respective interfaces are also shown in Listing 6.6 – the role class, for instance, encapsulates the pairing of a role name and an artefact stub, which is essentially an artefact without type information.

XML annotation attributes are used in most classes to control how the class's properties are rendered to XML. The most important attributes are XmlAttribute, which declares that the property is to be rendered as an attribute of the element representing the class instead of the default – nested element – and XmlIgnore, which tells the XML serializer not to render the property at all. Also useful is XmlText, which makes the property's value the text contained in the current element, dropping the property name; this is used in the Role, ArtefactType and RelationshipType classes to make the specific type the text content of the Type element, while the generic type is rendered as attribute.

Despite leaning towards XML serialization, the component provides means to control the serialization method to use via SerializerFactory. XmlMessageSerializer is, however, the only implementation of the corresponding interface, but the generic interface – it only has two methods to convert a message to a stream and vice versa – makes it easy to implement a serialization method for messages in a semantic web language, for example.

```
public static class MessageFactory
{
   public static IArtefactMessage CreateArtefactMessage(IArtefact artefact,
                                       ArtefactAction action, Sender sender)
   {
   return new ArtefactMessage(sender, new Artefact(artefact), action);
   }

   public static IArtefactArtefactRelationshipMessage CreateRelationshipMessage(
                               IArtefactArtefactRelationship relationship,
                               RelationshipAction action, Sender sender)
   {
   return new ArtefactArtefactRelationshipMessage(sender,
                               new ArtefactArtefactRelationship(relationship), action);
   }
}

public interface IArtefactStub
{
   string ID { get; }
   int Revision { get; }
}

public enum ArtefactGenericType{ Document, Report, Model, ModelElement, Diagram,
      Record, SourceFile, BinaryFile, ConfigurationFile }

public interface IArtefact : IArtefactStub
{
   string Name { get; }
   ArtefactGenericType GenericType { get; }
   string SpecificType { get; }
}

public enum ArtefactArtefactRelationshipType{ Aggregation, Dependency,
        Realization, Association}

public interface IRole
{
   string Name { get; }
   IArtefactStub Artefact{get;}
}

public interface IArtefactArtefactRelationship
{
   ArtefactArtefactRelationshipType GenericType { get; }
   string SpecificType { get; }
   IRole RoleA { get; }
   IRole RoleB { get; }
}

public struct Sender
{
   [XmlAttribute]
   public string ClientID { get; set; }
   [XmlAttribute]
   public string UserID { get; set; }
}

public enum ArtefactAction { New, Modify, Delete }
public enum RelationshipAction { Create, Validate, Delete }
```

Listing 6.6: Message factory and input types

6.3.3 Adapting Requirements Manager to Send Messages

The first step in employing this messaging system is adapting a tool to send messages. In this case, Requirements Manager has been adapted to send information about its use cases, features and constraints – just the artefacts, no relationships at this point. Constraints, also called non-functional requirements, are rather difficult to include in a traceability concept, but should be included nonetheless [46]. In this implementation, they are treated like functional requirements (features), the conceptual problems are left to be worked out by the traceability process.

The business logic layer, i.e. the service classes managing the artefacts, is the best place to put the responsibility of sending messages about new, changed or deleted artefacts. While the persistence layer would have the necessary information about single artefacts, and sending messages about CRUD operations might even be seen as a kind of persistence, it does not provide the best outlook in regard to receiving, filtering and processing messages. Since sending and receiving are done by the common messaging client, these operations should be performed in the same layer. On the other end of the spectrum, the presentation layer should not be concerned with message sending – deciding when to send a message is clearly a "business rule".

That being said, the next design choice is how to incorporate the necessary interaction with the client into the service classes. Since there is only one communication channel shared by several service classes, it makes sense to build a separate class to handle all communication, and make that class a singleton. Listing 6.7 shows the outline of the MessageGateway class. Thread safety and locking of that resource is not an issue in this implementation because the application is single-threaded, with the exception of pushed message handling, which is done in its own thread, but again in one thread for all messages.

```
public class MessageGateway : IMessageReceiver, IExternalArtefactMessageGateway
{
   public static void Initialize(ConnectionStringSettings connectionString,
                                 string userID )
   public static MessageGateway Instance { get; }

   private Sender MessageSender { get; set; }
   private IClient MessagingClient{ get; set; }

   public void SendArtefactMessage(IInternalArtefact artefact,
                                   ArtefactAction action);

   private static string GetClientIPAddress();
}
```

Listing 6.7: MessageGateway class outline for sending artefact messages

In order to use the message gateway, it first has to be initialized – the Instance property is null initially, and only set after a call to Initialize(). This method will set the values of the two private properties: the MessageSender property contains the sender information for outgoing messages' headers, while the MessagingClient property contains the client object obtained from the common messaging component. In order to properly configure the messaging client for callbacks (message push), the IP address of the machine running the application has to be determined, which is what GetClientIPAddress() does.

The method for sending artefact messages takes an artefact and the action performed on it as parameters. The ArtefactAction enumerator is defined in the common messaging component. The type of the artefact parameter, on the other hand, is defined in the component used for managing relationships to artefacts of other tools, which is used by both Requirements Manager and Test Tracker and is discussed in a later section. It is the common parent interface for all artefacts the Requirements Manager wants to share.

Since IInternalArtefact does not derive from the IArtefact interface defined in the common messaging component, it cannot be passed to the messaging client directly. This would be the case with any existing tool that is adapted to participate in traceability messaging, at least initially. When adapting the tool, one way would be to change the domain model so that its objects do implement IArtefact. A less intrusive approach is to develop one or several wrapper classes that implement it and have them translate the application-internal objects to the message factory. Since one of the arguments for the message-based approach is the possibility of non-intrusive adaptation, this is how it was done here. A single ArtefactWrapper class suffices to translate use cases, features and constraints into artefact objects that the message factory can use.

With the MessageGateway and the ArtefactWrapper in place, sending a message from a service class can be done in a single line of code, as this example from the UseCaseService.Delete method shows:

```
MessageGateway.Instance.SendArtefactMessage(useCase, ArtefactAction.Delete);
```

The only other modifications necessary are a configuration section in the application configuration file that includes connectivity information (see Listing 6.8) and initialization of the message gateway at an appropriate time – in case of Requirements Manager, it is done when the application loads the main form.

```
<configuration>
  [...]
  <appSettings>
    <add key="MessageServerName" value="192.168.0.101"/>
    <add key="MessageServerPort" value="7607"/>
    <add key="MessageServerUsername" value="john.doe@example.com"/>
    <add key="MessageServerPassword" value=""/>
    <add key="MessageClientCallbackPort" value="7611"/>
  </appSettings>
  [...]
</configuration>
```

Listing 6.8: Application settings for the messaging client

6.3.4 Adapting Test Tracker to Receive Artefact Messages

With one application sending information about its artefacts, the next step is to make the other tool receive and process them. Not only that, but the received information has to be persisted by the recipient in order to be usable for traceability. Because the information available through messages is generic in nature, the prototype also persists it in a generic way. If tighter integration of a set of tools is desired and therefore extensions to the message format are made, storing different artefact types in different tables may become opportune, but this is not the case here.

Since the two tools share a common architecture, not only does it make sense to use the same message gateway and wrapper concept, but also to develop a shared component that handles artefacts external to the application in a generic way. By using that component and including the table shown in Figure 6.13 the application can be equipped to handle external artefacts.

Database Extensions

The traceability information received by messages has to be persisted in order to be usable. Tools may choose not to persist every information they receive, but in the prototype implementation, no such filtering takes place. Furthermore, a generic approach has been chosen to be able to handle all possible messages and demonstrate that even without artefact type-specific information, there is already significant benefit in processing messages and incorporating received information into the application.

In order to store the information received in artefact messages, the entity depicted in Figure 6.13 has been added to Test Tracker's database.

The ExternalArtefact table stores all relevant information about any artefact that the tool has become aware of by receiving a corresponding message. They type attribute only stores the specific type – at this point, there is no benefit in

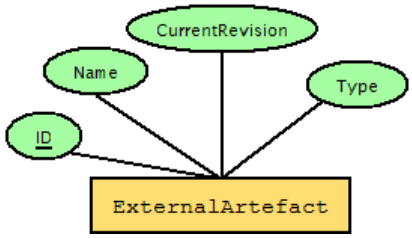

Figure 6.13: Additional Table for persisting external artefacts

persisting the generic type because artefacts are not treated differently according to their generic type (this may change with extensions). The CurrentRevision attribute contains the last revision associated with the artefact in a received message about the artefact. It is used to determine whether a relationship is suspect or not, as will be described later.

Building a Component for External Artefacts

The database schema extension for handling artefacts external to the tool whose database is extended, as well as relationships to these artefacts as discussed below, is the same in both tools. Therefore, a common component that accesses these additional tables and offers a domain model and methods to handle external artefacts and traceability links has been developed.

This component acts as a translator between the artefacts as described in the messages passed between the two tools and the internal representation of those artefact in the respective tool's database. Listing 6.9 shows the methods of the ExternalArtefactService class that deal with translating artefacts between representations.

```
public class ExternalArtefactService : BaseService
{
   public ExternalArtefactService(ConnectionStringSettings connectionString,
                                   IExternalArtefactMessageGateway messageGateway)

   public IExternalArtefact GetArtefact(IArtefact artefact);
   public IExternalArtefact GetArtefact(string artefactID);
   public List<IExternalArtefact> GetArtefactList();

   public void SaveArtefact(IExternalArtefact artefact);
   public void DeleteArtefact(IExternalArtefact artefact);
   [...]
}
```

Listing 6.9: ExternalArtefactService class for artefacts only

The GetArtefact methods both try to find the corresponding external artefact in

the database, build the domain model object (internal representation) from the data and return it. The first overload can also create a new domain model object from the message representation if the artefact is not found in the database, which can then be persisted by calling SaveArtefact.

Extending the MessageGateway Class

The MessageGateway class introduced in a previous section is the single point of messaging. In order to receive messages and process them, it has to be extended by the members highlighted in Listing 6.10 – a method to be called from the messaging client, which is an implementation of the IMessageReceiver interface, and an event that is fired after the message has been processed and the external artefact object created from its contents. The Receive method uses the ExternalArtefactService to get the existing internal representation or create a new one, and also to save the updated external artefact or delete it if the message calls for it.

```
public delegate void ArtefactMessageEvent(IExternalArtefact artefact);

public class MessageGateway : IMessageReceiver, IExternalArtefactMessageGateway
{
    public event ArtefactMessageEvent ArtefactMessageReceived;

    public static void Initialize(ConnectionStringSettings connectionString,
                                  string userID )
    public static MessageGateway Instance { get; }

    private ExternalArtefactService ExternalArtefactService { get; set; }
    private Sender MessageSender { get; set; }
    private IClient MessagingClient{ get; set; }

    public virtual void Receive(IArtefactMessage message);

    public void SendArtefactMessage(IInternalArtefact artefact,
                                    ArtefactAction action);

    private static string GetClientIPAddress();
}
```

Listing 6.10: MessageGateway class outline for receiving artefact messages

User Interface Extensions

In order to create value for the user of Test Tracker from the information about external artefacts, controls to establish links between test suites or test cases and those external artefacts have to be added. The most important addition in this respect is the introduction of a traceability tab in the detail view of both test suite and test case, which can be seen in Figure 6.14.

In this view, all relationships of the currently selected item – in this case, the test case “Navigate to next month” – to external artefacts are shown. The first

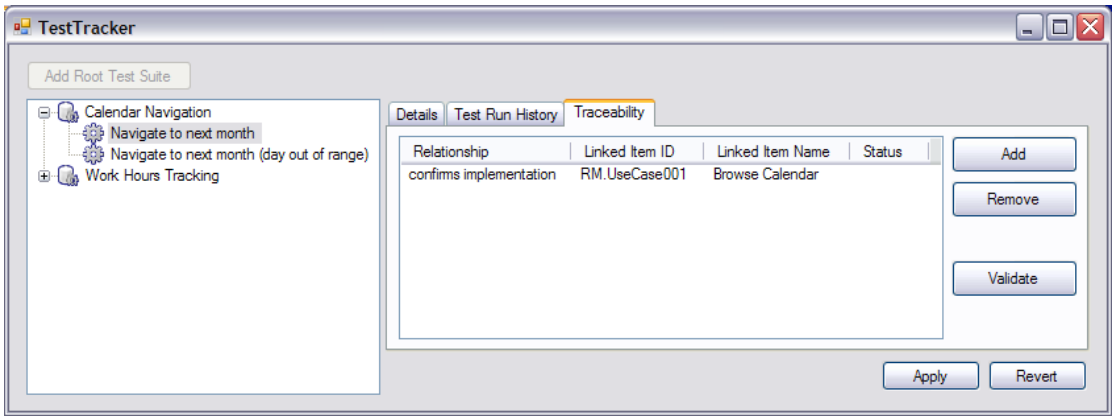

Figure 6.14: Traceability tab in Test Tracker

column shows the specific relationship type, the second and third columns show the ID and name of the external artefact as messaged by Requirements Manager, and the last column shows whether the relationship is suspect – an empty cell means it is not.

The add button opens up the dialogue in Figure 6.15, where the user can choose from a list of external artefacts the one to link to. The list features all artefacts that have been messaged to Test Tracker, so this list may become too big to handle in a real project. That is why the list can be filtered so that only artefacts of selected types are shown; the filter link on the upper rights leads to a corresponding dialogue (Figure 6.16) that shows all known external artefact types, which can be included by checking the check box next to the type name. New artefact types are included by default.

When adding a traceability link, a relationship type has to be entered. The corresponding combo box allows entering a new relationship type as well as choosing one that has already been used, as shown here. When a new type is entered, the combo boxes for test case role and external artefact role are of course editable.

The “Validate” button removes the “suspect” mark on the relationship and sends a corresponding relationship message of type “validate”, with the revision of the target use case equal to the last known revision. This means that the relationship will be marked suspect again when a new revision of the use case is messaged.

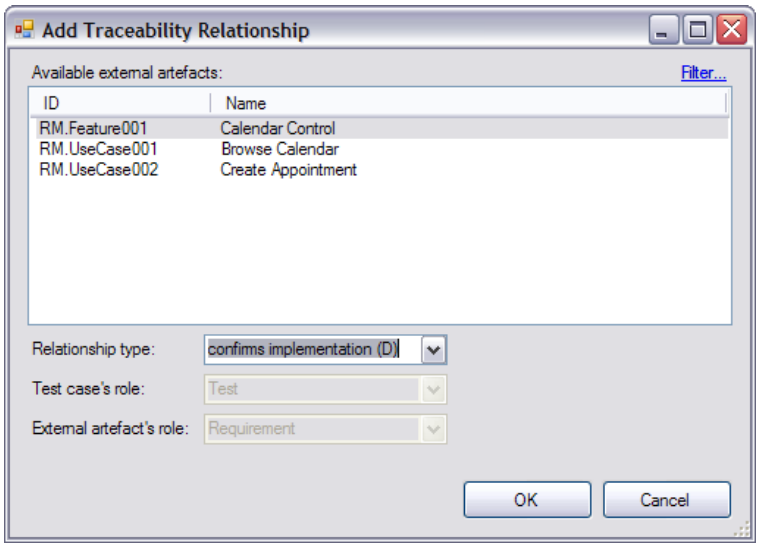

Figure 6.15: Adding a traceability link in Test Tracker

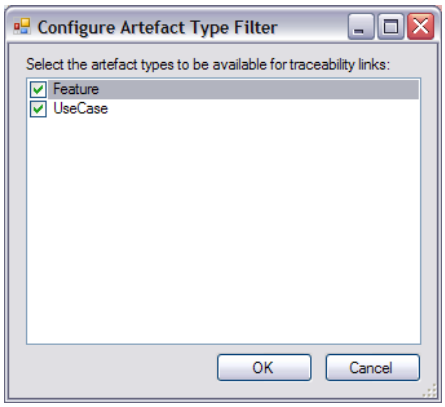

Figure 6.16: Filtering external artefacts by type

6.3.5 Adapting Test Tracker to Send Relationship Messages

Now that users can establish relationships between test cases and requirements in Test Tracker, the tool must be adapted to send information about those relationships to Requirements Manager so that it becomes aware of it. But first, Test Tracker needs to persist relationship information itself so that the information shown in the traceability tab does not get lost when exiting the application.

```
<configuration>
   <configSections>
      <section name="ArtefactTypes"
               type="TestTracker.Main.ArtefactFiltersConfigurationSection, TestTracker" />
   </configSections>
   [...]
   <ArtefactTypes>
      <add key="Use Case" value="true" />
      <add key="Feature" value="false" />
   </ArtefactTypes>
</configuration>
```

Listing 6.11: Configuration section for artefact type filter

There are three things to persist: relationships, their types, and the artefact type filter shown in Figure 6.16. The latter is an application configuration option an thus goes into the appropriate configuration file – the section reserved for this is shown in Listing 6.11 – while the former two require database extensions.

Database Extensions

In addition to the ExternalArtefact entity already discussed, there are two more entities needed to link artefacts managed by Requirements Manager (internal artefacts) and other tools, in this case Test Tracker (external artefacts) with a typed relationship. The complete set of database extensions can be seen in Figure 6.17.

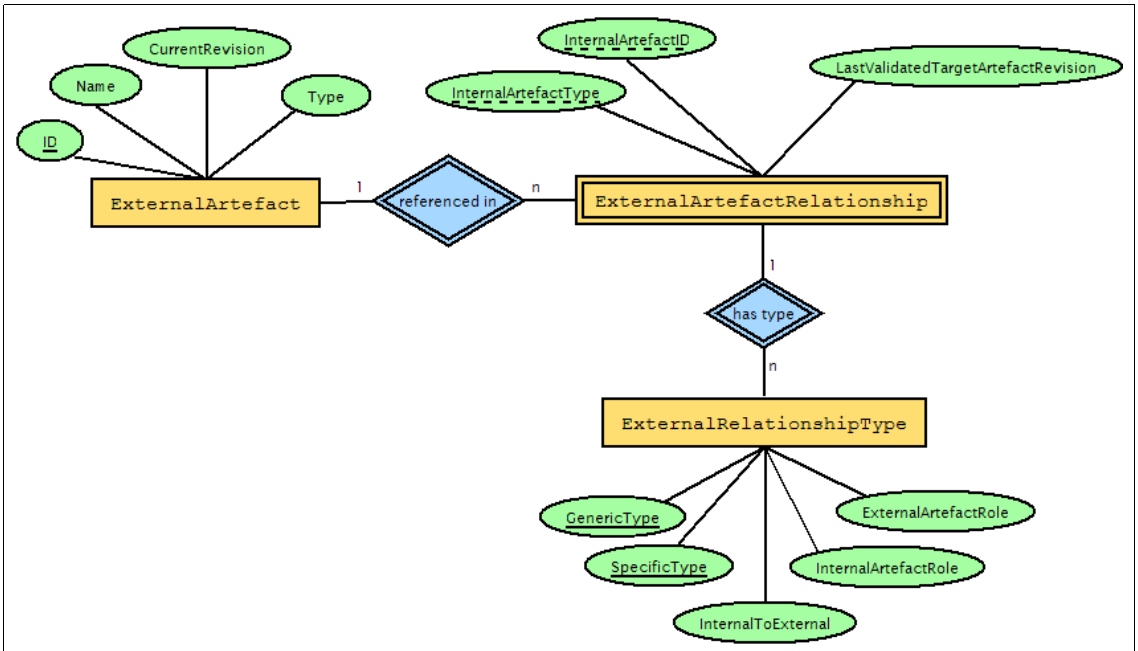

Figure 6.17: Additional tables for persisting traceability information

Relationship types are stored with their generic and specific parts, mainly because the combination has to be unique and thus makes a good primary key. The role attributes are optional text fields that help define the meaning relationship. For example, the “verifies implementation” relationship between test cases and use cases, features or constraints is a dependency relationship where the test case has the role “test” and the other artefact has the role “requirement”. While the meaning of the relationship may not be derived from only the relationship type name, with the role names it becomes obvious. The InternalToExternal attribute is a boolean value that indicates the direction of the relationship, in this case the dependency is from test case to requirement, and the value will be true in Test Tracker and false in Requirements Manager.

Lastly, the ExternalRelationship table connects internal artefacts – referenced by type and identifier (without relational integrity), the external artefact and a relationship type to express the traceability link between the referenced artefacts. The additional attribute LastValidatedTargetArtefactRevision is used for deriving the suspect status of the relationship: the target artefact is either the external or the internal artefact, as specified by the relationship type's InternalToExternal attribute; the relationship is suspect if the last validated revision is less than the target artefact's current revision. The value of this attribute can be changed by relationship messages of the type “validate”.

Extending the External Artefacts Components

The next step in making Test Tracker persist relationships to external artefacts is to add features to the external artefacts component that allow the application to create, edit and delete relationship types and relationships. Listing 6.12 highlights the new methods in the ExternalArtefactService class.

The principle of the GetRelationshipType and GetRelationship methods is the same as with the GetArtefact(IArtefact) method – if the relationship type or relationship is not in the database, a new one is created based on the data given as arguments. This is used when receiving a relationship message, which will be discussed in later sections; when creating a relationship in Test Tracker, the explicit CreateRelationship method is called.

```
public class ExternalArtefactService : BaseService
{
    public ExternalArtefactService(ConnectionStringSettings connectionString,
                                   IExternalArtefactMessageGateway messageGateway)

    public IExternalArtefact GetArtefact(IArtefact artefact);
    public IExternalArtefact GetArtefact(string artefactID);
    public List<IExternalArtefact> GetArtefactList();

    public void SaveArtefact(IExternalArtefact artefact);
    public void DeleteArtefact(IExternalArtefact artefact);

    public IRelationshipType GetRelationshipType(EnumGenericRelationshipType genericType,
                                   string specificType)
    public List<IRelationshipType> GetRelationshipTypeList()

    public IExternalRelationship GetRelationship(IArtefactArtefactRelationship relationship,
                                   IInternalArtefact internalArtefact,
                                   IExternalArtefact externalArtefact)
    public List<IExternalRelationship> GetRelationshipList(IInternalArtefact item);
    public IExternalRelationship CreateRelationship(IInternalArtefact item,
                                   IExternalArtefact artefact, IRelationshipType type)
    public void SaveRelationship(IInternalArtefact item, IExternalRelationship relationship)
    public void DeleteRelationship(IExternalRelationship relationship)
}
```

Listing 6.12: ExternalArtefactService class – complete

Extending the MessageGateway Class

The single point of sending and receiving messages has to be extended to send relationship messages. The corresponding method takes an IExternalRelationship object and an enumerator to specify the relationship action, and is called from the ExternalArtefactService class when saving or deleting relationships.

6.3.6 Adapting Requirements Manager to Receive Relationship Messages

In order to be able to display traceability information (that has been generated in Test Tracker) in Requirements Manager, the tool has to be adapted to receive and process artefact and relationship messages, and store that information in its database. The prerequisites for doing so – database extensions and the common external artefacts component – have already been discussed in the previous sections, and need only be applied to Requirement Managers analogously. The things left to describe are the necessary additions to the MessageGateway class so that relationship messages can be processed, and the extensions to the user interface that show traceability information that has been received.

Extensions to the MessageGateway Class

First and foremost, a Receive method is needed for relationship messages; with this method, the implementation of the IMessageReceiver interface is complete and the messaging client can call the appropriate function when a message is

pushed. Since a relationship message does not explicitly state which tools the artefacts participating in the relationship belong to, there is a GetInternalArtefact method that finds out which of the two artefacts, if any, is the artefact native to the current tool, and returns it. In order to do so, it needs one of the three business logic service classes to retrieve the actual object.

```
public class MessageGateway : IMessageReceiver, IExternalArtefactMessageGateway
{
    public event ArtefactMessageEvent ArtefactMessageReceived;
    public event RelationshipMessageEvent RelationshipMessageReceived;

    public static void Initialize(ConnectionStringSettings connectionString,
                                  string userID )
    public static MessageGateway Instance { get; }

    private UseCaseService UseCaseService { get; set; }
    private FeatureService FeatureService { get; set; }
    private ConstraintService ConstraintService { get; set; }
    private ExternalArtefactService ExternalArtefactService { get; set; }
    private Sender MessageSender {  get; set; }
    private IClient MessagingClient{ get; set; }

    public virtual void Receive(IArtefactMessage message);
    public virtual void Receive(IArtefactArtefactRelationshipMessage message);

    public void FetchMessages();
    public void SendArtefactMessage(IInternalArtefact artefact,
                                  ArtefactAction action);
    public void SendRelationshipMessage(IExternalRelationship relationship,
                                  RelationshipAction action);

    private static string GetClientIPAddress();
    private IInternalArtefact GetInternalArtefact(IArtefactStub artefact)
}
```

Listing 6.13: MessageGateway class outline - complete

The ExternalArtefactService is used to fetch the existing relationship from the database or create a new one if it is yet unknown. Depending on the artefact roles contained in the message, the Receive method can determine if the relationship is directed from the internal to the external artefact (internal has role A, external has role B) or vice versa.

User Interface Extensions

In this prototype, Requirements Manager only displays traceability information generated by Test Tracker, but does not generate relationship messages on its own. Like in Test Tracker, there is a traceability tab in the detail view, as shown in Figure 6.18 – it has only been implemented for use cases, but the infrastructure and data for providing the same view for features and constraints is available.

In the screen shot, the second relationship is marked “suspect”, which means that since it has been established or last validated, the target artefact – in this

case, the use case “Browse Calendar” – has changed. The revision number of the artefact is thus higher than that received in the last message concerning that relationship. Validating the relationship in Test Tracker issues a message that will update the relationship in Requirements Manager, thus removing the suspect flag. Since the test case depends on the use case, it would not make sense to enable validating from Requirements Manager, because after a use case has been changed, the dependent test case has to be reviewed for necessary adaptations.

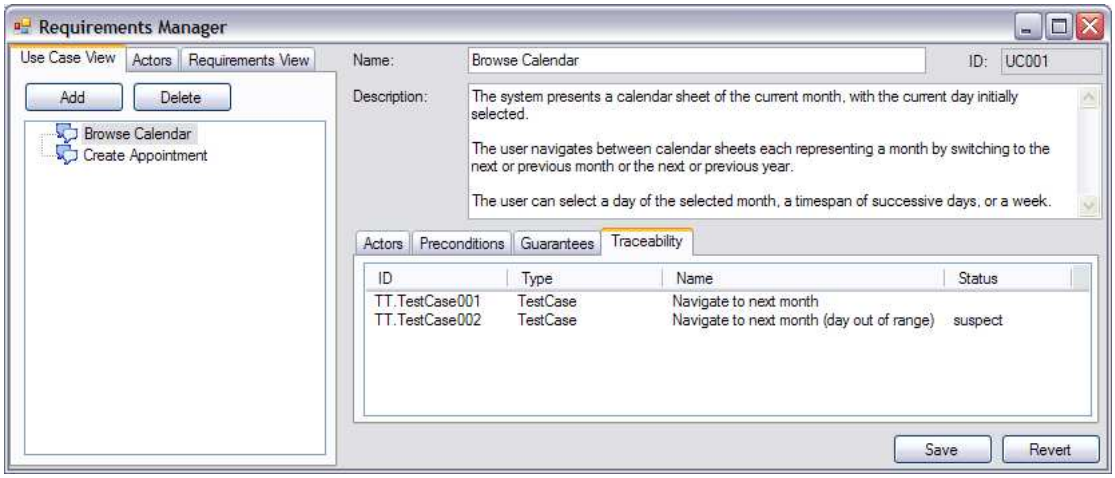

Figure 6.18: Traceability tab in Requirements Manager

6.4 Results

The prototype consists of two tools, Requirements Manager and Test Tracker, that have been developed as stand-alone tools and adapted to communicate through messages of the format presented in chapter 6. Requirements Manager features traceability between internal artefacts, namely use cases and features/constraints to contrast the implementation of internal traceability links with that of links to or from artefacts managed by other tools.

When a use case, feature or constraint is created in Requirements Manager, a message is issued and received by Test Tracker, which stores the delivered information in a generic database schema extension. The knowledge about Requirements Manager's artefacts enables Test Tracker to establish traceability links to these artefacts. Test Tracker's user interface has been extended to include dialogues and controls that can add relationships to external artefacts it knows about, and add relationship types on the fly.

Test Tracker then sends information about the newly established relationship to

Requirements Manager, who stores that information in the same generic database extension, which makes the traceability link bi-directional [47]. Requirements Manager displays the traceability links targeting its use cases in a dedicated tab on the use case's detail view. When the use case is updated, a message about the new revision is sent to Test Tracker, making the relationship suspect because it is a dependency relationship where the target artefact (the dependee) has changed, wherefore the source artefact (the dependent) has to be reviewed for need of adaptations.

When the test case has been reviewed and adapted if necessary, a user of Test Tracker can validate the relationship, thus removing the suspect flag and sending a message about the validation to Requirements Manager so that the relationship can be updated there, too.

While putting this communication into place took considerable effort, it did not require significant changes to the tools themselves, but instead could be implemented by additional classes within the tools' business logic and an external component that had to be referenced and called from a few existing methods. The necessary database extensions also did not touch the existing schema but only added tables, neither of which has direct relationships with existing ones.

The only major change required was to introduce revisions to the tools' artefacts. This was needed for demonstration of the “suspect” mechanism for relationships. However, if a candidate tool for adaptation does not have revisions for its artefacts, it can always report revision number 1, for example. It could have been done here in Test Tracker even without sacrificing the “suspect” mechanism, because test cases are not target artefacts of dependencies in this prototype. Since version control is becoming more and more prevalent even for artefacts not classifying as source code, candidate tools not keeping track of their artefacts' revisions will become fewer. Furthermore, since a revision number only has to be greater than its predecessor, it could be derived from a status, a time stamp or the like instead of explicitly tracking revisions.

Using the guidelines defined in [24] to judge the viability of the solution, it can be said that they are all fulfilled. Costs are minimized as existing tools can be adapted with relatively little effort, the problem space is be bounded since only select artefact types are traced, existing work practices are supported by adapting tools that are already in use, information is only entered once and transferred via messages, and only necessary automation is done (suspect flags).

7 Other Approaches for Tool Integration

When introducing a new concept, it is always advisable to look at solutions that already exist. However, in the case of enabling traceability by integration of existing tools, there is next to nothing to be found.

One effort that did achieve workable results was the OPHELIA project, which was sponsored by the European Union. The open source community, as represented by sourceforge.org, has little to show in this regard, or in regard of integrated development in general. Companies usually only integrate their own products with each other, or sometimes their own products with very few select others – an approach that can be termed “proprietary integration”. Another approach altogether is integrating different tools into one platform: the Eclipse project.

7.1 Proprietary Tool Integration

One of the first software vendors to offer a suite of software engineering tools that were integrated to some extent that allowed cross-tool traceability was Rational, now an IBM division. Their products RequisitePro and Rose – a requirements management and UML modelling tool, respectively – were integrated so that use cases in Rose could trace to requirements, e.g. features, in RequisitePro [48]. This example shows the primary problems with proprietary tool integration: it is limited to certain artefact types.

Although there are currently more systems available that offer traceability between their artefacts, the fundamental problem with these is that they are either closed off or offer a proprietary API. When an API is available, integration can be added for each pair of tools, which is rather cumbersome even if not all $n(n-1)/2$ communication channels are required – a case study for Rose and DOORS [49], a requirements management software from a competing vendor, is given in [50]. While this approach allows for tight, customized integration, it requires an enormous effort compared to a central repository approach or the messaging system presented in this thesis.

A more general solution is offered by elego Software Solutions [51]: the project's artefacts and their interrelations organised in a tool-independent meta-model, which is stored in a common source control repository. This reduces complexity, because now each tool has to be integrated only with the common meta-model. The approach is conceptually similar to that of OPHELIA: build a central repository and connect the tools to that repository, requiring tools to adapt to the concepts put forth by the meta-model. Since the meta-model is more specific than that used in the messaging system (artefacts with generic types), tool integration can be tighter, but tool developers potentially face problems when

matching their tools' concepts to those of the meta-model.

7.2 Platform Integration – Eclipse

Although the name Eclipse is mostly associated with a Java IDE, the Eclipse platform has actually been conceived as an open development platform, which manifests in its powerful plug-in mechanisms. The idea of integrating all tools needed during a software development project into the Eclipse client seems obvious, especially since the back-ends of existing applications can be reused and only the user interface has to be reimplemented as Eclipse plug-in that connects to that back-end. This has been demonstrated by IBM Rational, whose defect-tracking product ClearQuest is available as native Windows and Unix clients, web client or Eclipse plug-in [52].

However, having all the applications available as Eclipse plug-ins does not mean that they are aware of each other or that they can share data. Plug-ins have to explicitly reference other plug-ins, listing them as prerequisites or optional references in their manifests. This means that the problem of establishing up to n(n-1)/2 communication channels still needs to be addressed or a central repository be established. It is therefore not a tool integration in the sense that it enables traceability, but only integrates user interfaces into a single client application – traceability integration can be added, but it requires about the same effort as without integration in Eclipse.

Another effort in platform integration – the Software Concordance Editor – is described in [53]. In this case, traceability is supported through hyper-links in documents, but any non-textual artefacts are left out.

7.3 The OPHELIA Project

The "Open Platform and metHodologies for devELopment tools IntegrAtion in a distributed environment" project [54] ran from October 2001 to December 2003, starting out on the same premises as this theses: traceability needs tool integration, commercial suites exist but are too expensive for small teams and community projects, the open source community has not produced satisfactory solutions themselves. A collaboration of academia and industry, partly funded by the European Union, set out to create a platform to which existing tools could be connected. The idea was to create an abstraction layer for existing tools and a central repository to store traceability information in.

7.3.1 Integration Approach

The abstraction layer consisted of ten CORBA interfaces for modules such as requirements, modelling or project. Applications had to implement at least one

of these interfaces to communicate with the repository. The idea was that a plug-in to the application would expose its functionality via that CORBA interface. That required some significant adaptation, as each application was required to run in client-server mode, thus requiring a server component to be written for desktop tools. Web-based tools needed an adapter to access the tool's database through the module interface.

Since the interfaces were designed to allow fairly tight integration, requiring tools to fulfil the whole interface contract also would have resulted in significant effort to bring the application in line with the module interfaces fitting the application's functionality. The benefit of this integration is that automatic or semi-automatic generation of new artefacts from existing ones is possible, as well as implementing autonomous agents that synchronize artefact's contents. In addition, the notification interface enabled changes to artefacts to create notifications to people responsible for dependent artefacts.

7.3.2 Traceability

Traceability was viewed as a special case of project meta-data, along with versions, physical location and other data. The abstraction of project artefacts to generic CORBA interfaces allowed traceability relationships between any two objects to be implemented in the so-called relationships layer. It thereby enabled relationship-oriented navigation and aforementioned notifications and synchronisation [55].

Another thought was to analyse the traceability graph to find areas of high coupling to deal with them before they became problematic to maintain, or to calculation project metrics. The key idea was that although the artefacts themselves would still be created and managed by the proper application, they could be abstracted sufficiently to track their traceability relationships in a common layer, making that information independent of any single tool and available for processing and analysis. However, putting all traceability links together in one repository tended to clutter the traceability graph [56].

7.3.3 Results

OPHELIA produced a suite of tools that implemented the interfaces defined by the project. While most have been developed for the project, four existing applications have been integrated, namely Microsoft Project, Bugzilla, CVS and ArgoUML. The idea was to let this suite serve as example for customized Ophelia solutions, which could include any commercial or open source application.

Although the project seemed promising and introduced many ideas that are

fundamental to the integration of software development tools, especially in regard to traceability, it has to be concluded that the immediate results did not have a lasting impact on the tool landscape. As of October 2008, Google search results do not indicate ongoing efforts to make tool OPHELIA-compliant, and the project's domain, www.opheliadev.org, is up for sale. Speculations on possible reasons for this apparent failure are given in chapter 6.

7.4 Meta-Modelling Approaches

In contrast to OPHELIA, this concept does not require adaptation of tools to a set of defined interfaces, but instead creates meta-models of the artefacts managed by the tools in a common language. Traceability links can then be established between instances of meta-model elements in a central repository. This approach has been described from different viewpoints in [57] (separation of concerns), [58] (mapping tools to meta-models), [59] (meta-model with topic maps) and [60] (meta-modelling based on the Meta Object Facility standard).

While the meta-modelling approach avoids forcing tools into a pre-defined conceptual model, it means that a meta-model has to be created for each tool to integrate, thus partially negating the advantage of reduced complexity achieved by OPHELIA's interfaces.

7.5 Community projects

Since developing software in their spare time is what the members of the open source development community are known for, it could be expected that they want to use that time efficiently. This hypothesis is supported by the seemingly countless projects creating tools to improve on the implementation process – tools for automatic unit testing, bug trackers and version control systems are amongst the most prominent and successful open source products supporting software development itself, whether as add-ins to integrated development environments (IDEs) or stand-alone tools.

In this section, the focus is on open-source efforts to integrate the whole development cycle from requirements gathering and management to implementation and testing. There are surprisingly few such projects to be found (a search for “traceability” gave 13 results on sourceforge.org on October 5th 2008, not all of which are related to traceability in software development), and those that do exist seem to try it all on their own, instead of integrating successful solutions to specific areas into a consistent whole.

7.5.1 Open Source Requirements Management Tool

This Java-based project by two developers created a rich client application that

functions as requirements and test management tool with traceability features. Additionally, the user can create proxies for implementation artefacts that point to source code files, thus integrating these artefacts into the tool's database and making them traceable. However, manually creating artefact proxies and referencing source files without a “find file” dialogue is rather impractical. Although design artefact proxies can be created, there seems to be no way to create models and diagrams, or to import or reference them.

The OSRMT project's latest release is 1.50, making its status on sourceforge.org “stable/production”. However, a short test suggests that this product is not quite finished – creating a test case with steps resulted in exceptions and could not be completed, for example. Furthermore, the the release is from March 2007 and the subversion repository has not seen a write transaction in the twelve months prior to October 2008. It seems as if the product is not being maintained any more, at least not by its original developers; the forums indicate that others have taken up the code and modified it, but not in an organised fashion.

7.5.2 COCONUT

COCONUT (COde COmprehension Nurtrant Using Traceability) is an Eclipse plug-in that enables that IDE to manage an artefact space – a collection of high-level artefacts such as use cases – and extract information from the artefacts to suggest names for classes and other identifiers while implementing. Although this may be a useful functionality, it does not really constitute a traceability solution, because any connection between use-case and implementation created by that tool is implicit in the name, not explicit in some sort of link. This loose connection does not allow for any automation using traceability links.

This project is in status “beta” with its release 1.1, but like OSRMT has not seen development activity from its four registered developers in the past year. The forums are also empty, so it seems that this product has been abandoned.

7.5.3 SLAM Software Lifecycle Artefact Manager

From the project statement and available on-line information, this effort aims to produce a Java application that uses OpenOffice Writer to edit requirements, and provide controls to manage test cases and test executions, while maintaining traceability between these artefacts. It does not try to include any modelling or implementation artefacts, nor does it offer any project management features.

The project has been registered in April 2008, but the Subversion repository has seen its first write transactions in September. The single developer claims his product to be in “beta” and has released a version 0.22, which failed to run

because it could not query the OpenOffice version, although version 2.1 was installed on the same system. Screen shots indicate, however, that significant progress has been made: requirements, test case and test execution management seem to be working. This project is rather likely to produce a functional requirements and test management tool with traceability in the near future, although with one developer and a yet non-existing user community, it is always in danger of being abandoned without notice.

7.5.4 Other Projects

Sourceforge has some other projects containing "traceability" in their description, but these are either not software development tools, limited to requirements only, or have not produced a workable product yet. Ignoring non-CASE projects and those limited to requirements artefacts, there are some approaches and project statements that would seem interesting but for their lack of results and activity. Below are the project statements and the current project status.

ETrace Tool

"ETraceTool is a traceability platform encapsulated in an Eclipse plugin. It allows the user to automatically trace a Java/EMF transformation. It also provides a graphic representation of the generated trace."

The project has been registered in August 2008 and has seen a lot of activity from its single developer in that month, but one in September. The official status is "alpha", and generated diagram is available in the screen shot area, although if it is an actual screen shot or rather a concept is unclear. There are no downloadable packages yet, but the Subversion repository already contains over 200 files. At this point, it cannot be predicted what the outcome of this effort will be.

Requirement Tracer

"A complete requirement traceability solution, which allows you to trace a requirement to the code entity level and generate an Excel report. This suite also provides plug-in to allow it to be embedded in to the automated build process."

This project has not submitted any files to its CVS repository since its registration in December 2004. There are no forum entries, and only a single registered developer. The project must be assumed dead, especially since the developer is not registered on any other projects, which could indicate a merger.

Requirements Manager

"A tool for Requirements Engineering, Management, and Development.

Possible features: process template support, stakeholder management, prototype integrations, use cases, requirements reuse, source-code & design dependencies, surveys, and traceability."

The project has been registered in the year 2000, and is officially still in the planning stage. It seems safe to say that there are no results to be expected.

Tema

"Tema is an integrated, browser based test management tool. It integrates requirements, test planning, test execution and defect tracking into a single repository than provides traceability and reporting within and across entities."

The official status is "stable/production", but there are no files in the CVS repository, no screen shots and no download packages; the forums are empty. The project has been registered in May 2007 and still has a single developer who is only on this project – there is little hope for seeing results.

8 Conclusion

This thesis aimed at providing an introduction to the traceability concept and its applications and benefits in software engineering, point out the problems in acceptance of current implementations and suggest a message-based tool integration as a new approach. A prototype implementation proved that the concept is viable and that existing applications can be adapted with a minimum of modifications to the existing code base.

8.1 Results

This thesis presents a solution to the problem of tool integration for traceability. In small software development teams, usage of tools from different vendors or open source projects is the norm, so there are no traceability links between artefacts created and managed by different tools. Requirements, models and model elements, classes, test cases and other artefacts should, however, be linkable in order to keep their contents consistent and to obtain an overview of the project's progress.

In contrast to the approach prevalent in proprietary tool suites, where traceability is enabled between certain types of artefacts, but not universally, the proposed message-based system can deal with any artefact and any relationship between two artefacts. This comes at the cost of losing tight integration potential – there may be considerable value in sharing more information between a given pair of tools than the messaging system can transport. This drawback is somewhat alleviated by using an extensible message format that allows for additional information about an artefact that other tools can simply ignore.

The messaging system also differs from the integration concept underlying OPHELIA, where a central repository has knowledge about all artefacts and contains the traceability links between them. The proposed system uses the tools' existing persistence capabilities, which can be easily extended to store information received via traceability messages. Knowledge is thus decentralized and cannot be accessed for reporting purposes unless the reporting tool (or another tool whose database the reporting tool can access) has been subscribed to the message distribution mechanism and has collected that information. This drawback is balanced by the fact that little or no additional infrastructure like dedicated servers is necessary to use the messaging system.

The meta-model chosen for the presented solution is deliberately generic. This is also a major difference to OPHELIA and other repository-based solutions – they establish a more specific meta-model to enable tight integration, thereby forcing tools to adapt to the repository's concepts in order to participate. As the

prototype implementation shows, significant benefits can be reaped even with this generic meta-model.

The requirements management and test tracking tools were developed for this thesis, yet their initial development focused on them as stand-alone tools. This was done so that the modifications necessary for integration through the messaging system could be considered separately. The results are encouraging: while it was necessary to extend the tools' databases and user interfaces, the messaging and logic associated with external artefacts could be handled by common components; only minor adaptations to the business logic were needed to actually trigger message sending.

While the user interface extensions were enough to show the available information, no attempts have been made to implement reports on test coverage or test results for the available requirements (termed requirements coverage views in [61]). However, with the information available in Requirements Manager, the former could be realized (test results are only available in Test Tracker). In Test Tracker, both kinds of reports could be done, since the project's requirements and their dependent test cases are known as well as the test result history of each test case.

The messaging system has thus been proven a viable option for the integration of existing tools and despite its generic meta-model, it is powerful enough to provide significant value through the enabled integration.

8.2 Future Work

The messaging system and its implementation presented here only serve to demonstrate a way to enable artefact traceability among existing tools that are not integrated with each other. Two directions of follow-up work can be envisioned: to bring artefact traceability to popular open-source and extensible closed-source tools, and to expand the messaging system to encompass activities and workers to become a complete process traceability solution.

The first task is to spread awareness of the traceability concept and its applications by selecting popular tools from all disciplines and – one by one – adapt them to send and receive traceability messages. Priority should be given to implementation environments, for the simple reason that every project uses an IDE, and that market concentration seems quite high. In addition, integration with a version control system could solve the problem when to send a message about an artefact creation or change (when it is checked in) and also provide revision numbers for the artefacts in that version control system.

The next step should probably be oriented towards unit testing frameworks, bug

trackers and other tools used by developers themselves. Only when the concept is introduced to developers as something that will help them get their work done will it find acceptance and meaningful applications in project management. When this is achieved, integration of tools at the fringe of development – like modelling and testing tools are the logical candidates for integration, while requirements, project management and deployment tools should probably be considered last. This is not to say that these are not important disciplines, only that software developers are more concerned with their immediate work and its interfaces, and that an initial focus on developers seems a key factor for success, considering that most projects on sourceforge.org for software development are for tools that help developers.

An important factor will be the gains in productivity and convenience for developers. Enabling navigation to linked artefacts would probably be the “killer application” that could establish the concept. Pop-ups showing partial content of artefacts used in the current activity may also prove efficient to that end. Another idea is automatic generation of traceability diagrams. Finding applications of traceability information that make developers' lives easier is essential to the success of the concept.

Existing obstacles and hindrances also have to be addressed. The most pressing of them is the message distribution system: without an easily set-up and conveniently configured message bus, there is little hope of gaining the necessary acceptance. Whether XML Blaster can be packaged for easy installation, other systems are better fits (e.g. email) or a system has to be developed from scratch is subject to research.

In parallel to popularizing the idea, it should be expanded to encompass workers and activities. This should prove necessary when project management tools are considered, which generate activity descriptions as artefacts and operate on worker proxies, assigning them activities. Resolving the confusing relationship between the project management artefacts and the fact that they represent the project model is a prerequisite for that.

As mentioned above, version control should be embraced early in the process of introducing the concept. These tools also have a special status, as they operate on different versions of artefacts managed in other tools. The messages presented in chapter 6 cannot handle communication between, for example, an IDE and the version control system that keeps track of the source files; messages of that kind are not even necessary, because there are no explicit links between an artefact and its most current version, only the revision number – that number should somehow correlate to how the version control system refers to artefact versions. For example, Subversion only has a revision number for a

whole repository – when a set of artefacts is checked in, that change set causes a new revision. An artefact belonging to the change set can have its revision number (in traceability messages) set to the new revision number of the repository, which will in many cases cause several numbers to be skipped when an artefact that has been stable over the last revisions is changed. This is not a problem, since revision numbers only have to increase at each change, but not necessarily by one.

In conclusion, the presented messaging system is a starting point. It does not provide value on its own, but only establishes a focus for integration efforts of existing and future tools. The first tool adaptations have to come from its advocates and propagators, probably in conjunction with the communities already maintaining the respective tool, and target software developers in areas where it helps them significantly, so that a critical mass can be reached.

Appendix: XML Schemas for Messages

XML Schemas are provided for both message types presented in this thesis, so that messaging adaptations can be tested for conformance with the specification.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://inso.tuwien.ac.at/Traceability/ArtefactMessage.xsd"
    elementFormDefault="qualified"
    xmlns="http://inso.tuwien.ac.at/Traceability/ArtefactMessage.xsd"
    xmlns:mstns="http://inso.tuwien.ac.at/Traceability/ArtefactMessage.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:complexType id="ArtefactType" name="ArtefactType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="generic">
          <xs:simpleType id="ArtefactGenericType">
            <xs:restriction base="xs:string">
              <xs:enumeration value="Document" />
              <xs:enumeration value="Report" />
              <xs:enumeration value="Model" />
              <xs:enumeration value="ModelElement" />
              <xs:enumeration value="Diagram" />
              <xs:enumeration value="Record" />
              <xs:enumeration value="SourceFile" />
              <xs:enumeration value="BinaryFile" />
              <xs:enumeration value="ConfigurationFile" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType id="Action" name="Action">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Create" />
      <xs:enumeration value="Modify" />
      <xs:enumeration value="Delete" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="ArtefactMessage">
    <xs:complexType>
      <xs:sequence id="MessageContentSequence">
        <xs:element name="Sender" minOccurs="1" maxOccurs="1">
          <xs:complexType mixed="false">
            <xs:attribute name="clientId" type="xs:string" use="required" />
            <xs:attribute name="userId" type="xs:string" use="optional" />
          </xs:complexType>
        </xs:element>
        <xs:element name="Timestamp" type="xs:dateTime" />
        <xs:element name="Artefact">
          <xs:complexType id="Artefact">
            <xs:sequence>
              <xs:element name="Type" type="ArtefactType" />
              <xs:element name="Name" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required" />
            <xs:attribute name="revision" type="xs:positiveInteger" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="action" type="Action" use="required" />
```

```
    </xs:complexType>
  </xs:element>
</xs:schema>

<?xml version="1.0" encoding="utf-8"?>
<xs:schema
targetNamespace="http://inso.tuwien.ac.at/Traceability/ArtefactArtefactRelationshipMessage.xsd"
    elementFormDefault="qualified"
    xmlns="http://inso.tuwien.ac.at/Traceability/ArtefactArtefactRelationshipMessage.xsd"
    xmlns:mstns="http://inso.tuwien.ac.at/Traceability/ArtefactArtefactRelationshipMessage.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:complexType id="RelationshipType" name="RelationshipType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="generic">
          <xs:simpleType id="RelationshipGenericType">
            <xs:restriction base="xs:string">
              <xs:enumeration value="Dependency" />
              <xs:enumeration value="Aggregation" />
              <xs:enumeration value="Realization" />
              <xs:enumeration value="Association" />
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType id="Action" name="Action">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Create" />
      <xs:enumeration value="Validate" />
      <xs:enumeration value="Delete" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="ArtefactArtefactRelationshipMessage">
    <xs:complexType>
      <xs:sequence id="MessageContentSequence">
        <xs:element name="Sender" minOccurs="1" maxOccurs="1">
          <xs:complexType mixed="false">
            <xs:attribute name="clientId" type="xs:string" use="required" />
            <xs:attribute name="userId" type="xs:string" use="optional" />
          </xs:complexType>
        </xs:element>
        <xs:element name="Timestamp" type="xs:dateTime" />
        <xs:element name="ArtefactArtefactRelationship">
          <xs:complexType id="Artefact">
            <xs:sequence>
              <xs:element name="Type" type="RelationshipType" />
              <xs:element name="RoleA">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Artefact">
                      <xs:complexType>
                        <xs:attribute name="id" type="xs:string" use="required" />
                        <xs:attribute name="revision" type="xs:positiveInteger"
                                use="optional" />
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="optional" />
                </xs:complexType>
              </xs:element>
              <xs:element name="RoleB">
                <xs:complexType>
```

```
                <xs:sequence>
                  <xs:element name="Artefact">
                    <xs:complexType>
                      <xs:attribute name="id" type="xs:string" use="required" />
                      <xs:attribute name="revision" type="xs:positiveInteger"
                              use="required" />
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string" use="optional" />
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="optional" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="action" type="Action" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Bibliography

[1] M. E. Fayad, M. Laitinen, R. P. Ward: Thinking Objectively: Software Engineering in the Small - *Communications of the ACM* vol. 43 pp. 115-118; 2000

[2] F. P. Brooks Jr.: The Mythical Man-Month (Anniversary Edition). Addison-Wesley; 1995

[3] Wikipedia: Traceability (2008/10/21), 2008, http://en.wikipedia.org/wiki/Traceability

[4] CMMI Product Team: Capability Maturity Model Integration (2008/11/01), 2002, http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr029.pdf

[5] W. W. Royce: Managing the development of large software systems: Concepts and techniques, *Proceedings of IEEE WESTCON*, 1987

[6] B. L. Kovitz: Practical Software Requirements. Manning; 1999

[7] P. C. Belford, A. F. Bond, D. G. Henderson, L. S. Sellers: Specifications a key to effective software development, *Proceedings of the 2nd international conference on Software engineering*, 1976

[8] E. Rang and K. Thelen: A Requirements to Test Tracking System (RTTS), *Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective*, 1985

[9] M. Jarke: Requirements Tracing, *Communications of the ACM*, 1998

[10] A. Finkelstein: Tracing Back From Requirements, *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, 1991

[11] O. C. Z. Gotel, C. W. Finkelstein: An Analysis of the Requirements Traceability Problem, *Proceedings of the First International Conference on Requirements Engineering*, 1994

[12] C. Fetters, T. Hsu, B. Smeed: Requirements tracing (2008/10/21), 1999, http://www.cis.ksu.edu/~hankley/d841/Fa99/chap2.htm

[13] L. Lavazza and G. Valetto: Enhancing Requirements and Change Management through Process Modelling and Measurement - vol. pp. 106 - 115; 2000

[14] P. Maeder, M. Riebisch, I. Philippow: Traceability for Managing Evolutionary Change (2008/11/01), 2006, http://www.patrickmaeder.de/pdf/SEDE06_maeder.pdf

[15] M. Jarke: The Nature of Requirements engineering. Shaker; 1999

[16] Object Management Group: Unified Modeling Language (2008/10/21), 2007, http://www.omg.org/technology/documents/formal/uml.htm

[17] Object Management Group: Object Constraint Language (2008/10/21), 2007, http://www.omg.org/technology/documents/formal/ocl.htm

[18] M. Riebisch and M. Hubner: Traceability-Driven Model Refinement for Test Case Generation, *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems* , 2005

[19] R. Watkins and M. Neal: Why and How of Requirements Tracing - *IEEE Software* vol. 11 pp. 104-106; 1994

[20] D. Leffingwell: Agile Requirements Methods (2008/11/01), 2002, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jul02/

AgileRequirementsJul02.pdf
[21] P. Reed: Transitioning from Requirements to Design (2008/10/21), 2002, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun02/TransitioningJun02.pdf
[22] S. Tessler and A. Barr: A Pilot Survey of Software Product Management (2008/11/01), 1996, http://www.stanford.edu/group/scip/avsgt/SEPG97paper.pdf
[23] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil: An Examination of Software Engineering Work Practices, *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, 1997
[24] H. U. Asuncion, F. Francois, R. N. Taylor: An End-To-End Industrial Software Traceability Tool - *Foundations of Software Engineering* vol. pp. 115-124; 2007
[25] A. Forward and T. C. Lethbridge: Software and Document Engineering: The relevance of software documentation, tools and technologies: a survey - vol. pp. ; 2002
[26] I. Sommerville and J. Ransom: An empirical study of industrial requirements engineering process assessment and improvement - vol. pp. ; 2005
[27] T. Addison and S. Vallabh: Controlling software project risks: an empirical study of methods used by experienced project managers - vol. pp. ; 2002
[28] R. N. Charette: Why Software Fails - *IEEE Spectrum online* vol. 42 pp. 42-49; 2005
[29] Standish Group: The CHAOS Report (1994) (2008/10/21), 1994, http://net.educause.edu/ir/library/pdf/NCP08083B.pdf
[30] K. Beck, C.Anres: Extreme Programming Explained. Addison-Wesley; 2004
[31] C. Lee, L. Guadango, X. Jia: An Agile Approach to Capturing Requirements and Traceability (), 2003, http://venus.cs.depaul.edu/xjia/tefse03.pdf
[32] B. Ramesh: Factors influencing requirements traceability practice - *Communications of the ACM* vol. 41 pp. 37-44; 1998
[33] J. Heumann: The Five Levels of Requirements Management Maturity (2008/11/01), 2003, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/feb03/ManagementMaturity_TheRationalEdge_Feb2003.pdf
[34] M. Riebisch: Supporting Evolutionary Development by Feature Models and Traceability Links, *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*, 2004
[35] S. P. Reiss: The Desert environment - *ACM Transactions on Software Engineering and Methodology (TOSEM)* vol. 8 pp. 297-342; 1999
[36] D. Garlan, R. Allen, J. Ockerbloom: Architectural Mismatch Or Why It's Hard To Build Systems Out Of Existing Parts, *Proceedings of the 17th International Conference on Software Engineering*, 1995
[37] J. Cleland-Huang, C. K. Chang, M. Christensen: Event-Based Traceability for Managing Evolutionary Change - *IEEE Transactions on Software Engineering* vol. 29 pp. 796-810; 2003
[38] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, Y. Shaham-Gafni: Model Traceability - *IBM Systems Journal* vol. 45 pp. 515-526; 2006

[39] K. Pohl: Process-Centered Requirements Engineering. John Wiley & Sons, Inc.; 1996
[40] B. Ramesh and M. Jarke: Toward Reference Models for Requirements Traceability - *IEEE Transactions on Software Engineering* vol. 27 pp. 58-93; 2001
[41] M. Heindl and S. Biffl: Risk Management with Enhanced Tracing ofRequirements Rationale in Highly Distributed Projects, *Proceedings of the 2006 international workshop on Global software development for the practitioner*, 2006
[42] P. V. Biron and A. Malhotra, Ashok: XML Schema Part 2: Datatypes Second Edition (2008/10/21), 2004, http://www.w3.org/TR/xmlschema-2/
[43] G. Cysneiros and A. Zisman: Traceability and Completeness Checking for Agent-Oriented Systems, *Proceedings of the 2008 ACM symposium on Applied computing*, 2008
[44] S. B. Palmer: Basic Semantic Web Language (2008/10/21), 2001, http://infomesh.net/2001/07/bswl/
[45] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Desgin Patterns. Addison-Wesley; 1995
[46] J. Cleland-Huang: Toward Improved Traceability ofNon-Functional Requirements, *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, 2005
[47] B. Ramesh and M. Edwards: Issues in the Development of a Requirements Traceability Model - vol. pp. 256-259; 1993
[48] Rational staff: Use Case Management with Rational Rose and Rational RequisitePro (2008/10/27), 2003, http://www.ibm.com/developerworks/rational/library/835.html
[49] G. Versteegen: Anforderungsmanagement. Springer; 2004
[50] L. K. Meisenbacher: The Challenges of Tool Integration for Requirements Engineering (2008/10/27), 2005, http://cui.unige.ch/db-research/SREP05/Papers/14.pdf
[51] Elego Software Solutions: Integration via meta model-based repositories (2008/10/21), 2008, http://www.elegosoft.com/en/solutions/tool-integration.html
[52] B. Krish and N.Le: Understanding the IBM Rational ClearQuest Client for Eclipse (2008/10/27), 2004, http://www.ibm.com/developerworks/rational/library/04/r-3089/
[53] E. V. Munson and T. N. Nguyen: Concordance, Conformance, Versions, and Traceability, *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, 2005
[54] M. Hapke, A. Jaszkiewicz et al.: OPHELIA—Open Platform for Distributed Software Development (2008/10/27), 2004, http://www.cs.put.poznan.pl/dweiss/site/publications/download/hapke-osic-2004.pdf
[55] M. Smith, D. Weiss, P. Wilcox, R. Dewar: The Ophelia Traceability Layer (2008/10/27), 2003, http://www.cs.put.poznan.pl/dweiss/site/publications/download/csmre-paper2.pdf
[56] K. Kowalczykiewicz and D. Weiss: Traceability: Taming uncontrolled change in

software development - *Foundations of Computing and Decision Sciences* vol. 27 pp. 239-248; 2002

[57] S. Herrmann and M. Mezini: PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments - *ACM SIGPLAN Notices* vol. 35 pp. 188-207; 2000

[58] P. Mason, K. Cosh, P. Vihakapirom: On Structuring Formal, Semi-Formal and Informal Data to Support Traceability in Systems Engineering Environments, *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, 2004

[59] J. Kelleher: A Reusable Traceability Framework using Patterns, *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, 2005

[60] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, A. Schürr: Metamodel-based Tool Integration with MOFLON, *Proceedings of the 30th international conference on Software engineering*, 2008

[61] M. Lormans, A. van Deursen: Reconstructing Requirements Coverage Views from Design and Test using Traceability Recovery via LSI, *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, 2005