INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Forschungs- und Lehreinheit I
Angewandte Softwaretechnik

Rationale-based Unified Software Engineering Model

Timo Wolf

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:	1.	Univ.-Prof. Bernd Bruegge, Ph.D.
	2.	Univ.-Prof. Dr. Barbara Paech, Ruprecht-Karls Universität Heidelberg

Die Dissertation wurde am 11.05.07 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.07.07 angenommen.

To Eva and Moritz.
– T.W.

Acknowledgements

I want to thank *Prof. Bernd Brügge, Ph.D.* for his great support, recommendations, and visions. This dissertation would not have been possible without him. I thank *Prof. Dr. Barbara Paech* for the long-term research collaboration. I am grateful to *Allen H. Dutoit, Ph.D.* who coached me during my research. I am thankful to all my colleagues from the Chair of Applied Software Engineering and in particular to *Monika Markl*, *Helma Schneider*, and *Uta Weber* for the organizational support. I want to thank my wife Eva and my son Moritz who saw me rarely while writing this dissertation.

Contents

Abstract

This dissertation addresses two problems of distributed software development projects: Inconsistencies between related artifacts such as documents and models and the inefficient collaboration of distributed project participants. In particular informal communication is reduced and project knowledge gets lost. The interaction and integration of existing methods and modeling tools are currently insufficient to address these problems.

We claim that a single consistent representation, supporting a unified data model for all documents and models including collaboration artifacts, helps to overcome these problems. In existing methods, collaboration artifacts are usually treated separately and maintained apart from the system model artifacts. As all artifacts are represented in one unified model, it is possible to implicitly capture and maintain artifact relationships as dependency traces. The dependency traces are used to support consistency of related artifacts. Informal collaboration artifacts, such as comments, and formal discussions for representing rationale, are externalized and also part of the unified data model.

This dissertation describes an extendable meta-model as a single consistent representation. Based on the meta-model we introduce the RUSE model, which integrates system models, collaboration models and organizational models. The meta-model and the RUSE model are realized in a tool called Sysiphus. Sysiphus was used in several academic and industrial software development projects as well as for teaching software engineering. Selected projects are used to evaluate the proposed concepts.

Diese Dissertation untersucht zwei große Probleme in verteilten Software-Entwicklungsprojekten: Inkonsistenzen zwischen abhängigen Entwicklungsartefakten wie zum Beispiel Dokumenten und Modellen, sowie die ineffiziente Kollaboration von verteilten Projektteilnehmern. Insbesondere ist die informelle Kommunikation reduziert und implizites Projektwissen geht verloren. Existierende Methoden und Modellierungswerkzeuge sind derzeit nicht ausreichend integriert um diese Probleme zu vollständig lösen.

Die Hypothese dieser Arbeit ist, dass eine uniforme Repräsentation eines vereinigten Datenmodells für alle Dokumente, Modelle und Kollaborationselemente zur Lösung dieser Probleme beiträgt. In existierenden Methoden werden die Kollaborationselemente in der Regel getrennt von der Entwicklung der Systemmodelle behandelt. Da in unserem Ansatz alle Artefakte in einem einzigen vereinigten Modell existieren, können Artefaktbeziehungen implizit im Model festgehalten und verwaltet werden. Bei auftretenden Änderungen können diese Beziehungen zur Identifikation von transitiv abhängigen Artefakten verwendet werden und Mechanismen zur Konsistenzerhaltung unterstützt werden. Informelle Kollaborationselemente, wie zum Beispiel Kommentare, und formale Diskussionsmodelle zur Repräsentation von Begründungen werden festgehalten und sind auch Teil des vereinigten Datenmodells.

Die Dissertation beschreibt ein erweiterbares Metamodell zur uniformen Repräsentation aller Artefakte. Darauf aufbauend wird das RUSE Model beschrieben, das ausgewählte Systemmodelle, Kollaborationsmodelle und Organisationsmodelle aus der Softwaretechnik integriert. Die Modelle sind in dem Werkzeug Sysiphus realisiert, das in einer Vielzahl von verschiedenen Projekten im akademischen und industriellen Kontext zur Evaluierung der Konzepte eingesetzt wurde.

Conventions

A sans serif typeface is used to highlight class names, literals and code passages. While it is customary to avoid spaces in class, object, operation and attribute names in source code—by beginning the following word with a capital letter right after the last letter of the preceding word (e.g. ModelElement)—this is unsuitable for typesetting in hyphenless justification. Therefore, we opted for a convention that is more natural for typesetting by introducing spaces between the words, which allowed for greater flexibility in automatic word wrapping. We hope that the sans serif typeface is sufficient to recognize the compound code-related names, such as in Model Element. Multiline code passages are an exception. They are set in a `constant width typeface` to preserve the original indentation, also eliminating the need for adding spaces just for typesetting.

Inline citations are accentuated by “double quotes.” Extensive citations are additionally set as blocks with a right and a left margin. Should the need arise for quotations within citations, we occasionally modified the quotes from “double” to ‘single’ without explicitly showing this change. All other changes to direct citations are marked within {curly brackets.}

CHAPTER 1

Introduction

Distributed software development is increasingly common in global companies. Organizational reasons include global mergers and acquisitions, which keep development organizations partitioned along the boundaries of the acquired companies. The claim is a benefit from the variations in labor costs of different geographical locations and customization and support at the client site. Organizations increase their pool of skilled and experienced employees. Native employees located in the countries of the clients enable the organizations to be more responsive and to have a shorter time to market [6, 40].

Beside the benefits, distributed development projects have to overcome many challenges to be successful. This dissertation addresses two challenges: Inconsistencies between related artifacts and inefficient collaboration of distributed project participants. Maintaining consistency between related artifacts such as documents and models that are developed by distributed project participants is even more difficult than in collocated projects. Efficient collaboration and communication is needed between distributed teams that develop related artifacts. Unfortunately, collaboration is also hindered by the distribution. In particular informal communication to overcome urgent issues is reduced. Missing cross-site project knowledge and awareness of activities and problems from teams located at foreign sites are problems.

1.1 Artifact inconsistencies

Software development projects have to deal with different activities that range from requirements elicitation, analysis, design, implementation and test to project management, change management, knowledge management, distributed communication and collaboration or maintenance management [19]. Depending on the project needs and the software engineering philosophy and experiences of the project managers, different techniques are used and different models and doc-

uments are created in these activities. For example in requirements elicitation, goals [2, 3], use cases [68, 93], user stories [7], or structured text are used to represent the requirements. The different techniques of the activities have different characteristics and currently there is no evidence which techniques are best capable for a successful project. But industry and research have recognized that the different activities and their outcomes are strongly interrelated and impact each other.

For example, the domain analysis is based and depends on the elicited requirements. The system design depends on the requirements, analysis, and defined design goals. System tests depend on the requirements while unit tests are based on the object design. When changes occur to any artifact, the change impact to related and depending artifacts needs to be analyzed, and impacted artifacts need to be changed accordingly.

Capturing dependency traces between related artifacts facilitates the change impact analysis. For example, a requirement can be linked to its analyzing models, the system design elements that realize the requirement and to the related system test cases. All related artifacts can directly be identified and reviewed when the requirement is the subject of a change. Moreover, the dependency traces can be used for quality analysis [9]. For instance, a requirement without any traces to the system design is probably not addressed. A requirement without traces to test cases cannot be tested easily.

Unfortunately, different specialized tools are used in software development projects to create and maintain the system models and documents. For example, requirements databases such as DOORS [121] or Rational RequisitePro [61] are used to capture and maintain requirements. UML case tools like Rational Software Architect [63], Rational Rose [62] or Together [15] are used to create the analysis, the system design and the object design. Testing tools such as Mercury TestDirector [83] are used to specify, execute, and collect the results of test cases. Word processing tools such as Microsoft Office Word [84] are used to aggregate and publish the developed models in activity related documents such as the requirements analysis document or the system design document [19]. Change management tools such as Rational ClearQuest [60] are used to control and manage changes across all different models and artifacts. Software configuration management tools such as Rational ClearCase [59] or Subversion [123] support artifact versioning and history access.

Each tool provides its own model and the integration of different tools is insufficiently supported, which hinders the interchange of models across tools. Integration solutions are based on export and import mechanisms, which lead to redundancies in the model representations. Maintaining consistency across redundant models is a problem for large models. The automated management of dependency traces across different artifacts and tools, as well as a quality analysis among all models is not supported. The models and documents are dispersed among different tools and media with the problem of fragile or no traces [12, 13].

Even if the dependency traces exist in one tool, or in a separated file like a spread sheet (e.g. Microsoft Excel), there is no single view that shows all traceability relationships and the related artifacts. Changing or deleting artifacts that are referenced by external dependency traces lead to inconsistencies or even to dangling links, which need to be manually maintained. Visualizing and tracing through all dependency traces is crucial for a complete change impact analysis and is needed to raise the awareness of artifact dependencies across distributed teams.

The problem gets worse, when the different artifacts are developed by distributed participants and the artifact representation media and tools are geographically separated. Effective communication and collaboration channels are needed to maintain consistencies across interdependent models.

1.2 Distributed collaboration

Researchers have noted the importance of communication and collaboration in software development [29, 51, 73, 102]. Curtis et al. [29], in a field study of several large projects, observed that documentation does not reduce the need for communication, in particular, during the early phases of the project, when stakeholders create informal communication networks to coordinate their representational conventions. They also observed that obstacles in informal communication such as organizational barriers and geographical distance can lead to misunderstandings in design conventions and rationale. Kraut and Streeter [73] note that formal communication (e.g., structured meetings, specifications, inspections) is useful for routine coordination while informal communication (e.g., hallway conversations, telephone calls, workshops) is needed in the face of uncertainty and unanticipated problems, which are typical of software development. They observed that the need for informal communication increases dramatically as the size and complexity of the software increases. Grinter et al. [51] studied several distributed projects that used different organizational models for coordination. They confirmed the findings of Kraut and Streeter about breakdowns in informal communication for the distributed case. Moreover, they found that the unequal distribution of participants and skills across sites and the difficulty in finding experts were recurring issues, independent of organization. Herbsleb et al. [53] found in a quantitative study of industry projects that cross-site work takes much longer than single-site work and requires more people. They also reported a strong relationship between the delay caused by cross-site work and the degree with which remote colleagues are perceived to help out when the workload is high.

A recurring theme in these studies is that informal communication is critical for rapidly disseminating implicit knowledge, in particular, with respect to change and unforeseen events. During informal conversations, participants learn of upcoming changes and the potential impact on their work. In global projects, informal communication is severely hampered, denying this informal knowledge

to its participants. When a change or an issue is raised through formal channels, the recipients are often surprised, they do not have access to its underlying rationale, and they need time to identify and get in touch with the relevant stakeholders. Given the effort required to communicate across sites, ambiguities are glossed over, messages are misunderstood, and more issues are introduced into the project. Under high workloads, the requests from other sites are treated with a lower priority than those coming from local participants. Relationships between sites become adversarial, which further hinders collaboration.

Berenbach [11] describes distributed complex projects where the requirements, the hazard analysis, and threat models are elicited by different geographically distributed teams or people with different skills. The compartmentalization of requirements gathering can be exacerbated on global projects in several ways. In the case of product lines and very large projects, requirements gathering may be broken up among regions and distributed among different business units. Traditional requirement management techniques tend to fail when the elicitation is distributed.

For example, some medical system product lines must be analyzed for potential hazards in order to be sold in certain countries. Once a product feature is associated with a hazard, requirements are created in order to mitigate the potential hazard. The requirements and the hazard analysis are created by teams that are responsible for the product of the related countries. Due to time pressure or organizational issues, the product teams for other countries have difficulties or skip reviewing these new requirements and do not analyze how the requirements impact their product. In the worst case, the different teams do not even know each other and cross-team relationships are overlooked [10].

1.3 Goal and approach

The hypothesis of this dissertation is that the problems described above can be addressed with an unified model that unifies the system models and documents with the collaboration artifacts and the organizational model of software development projects. By a single consistent representation that supports the unified model, the dependency traces between the artifacts are no longer fragile, but are, in fact, implicit in the unified model. Traces between system models and related collaboration artifacts can be implicitly captured and related stakeholders and experts can be identified as they are also part of the unified model.

This dissertation provides such a unified model, called rationale-based unified software engineering model (RUSE), which is defined as follows:

The **rationale-based unified software engineering model (RUSE)** integrates selected *system models*, the *collaboration model* and the *organizational model* of software development projects. It is based on an extendable meta-model that realizes shared requirements of the RUSE model. Traceability across all interrelated

models is supported and awareness among distributed project participants is increased. The RUSE model is designed to be located in a central repository that is shared across different distributes sites.

The *system models* contain all artifacts that describe the system under development. The kinds of artifacts depend on the applied process and range from features, requirements, use cases, design and object models, to dynamic models, test cases and even all written documents. The system models are usually based on UML[93] and are extended with new association classes that are used to capture traceability dependency links.

The *collaboration model* includes comments, an issue model, and work items. The comments are used for capturing informal discussion threads. The issue model is based on Question, Options, Criteria (QOC) [80] and facilitates the capturing of rationale and represent project knowledge. Work items describe the work to be done and can be assigned to the project participants. By integrating collaboration artifacts with the system models, we ingrain the communication channels in the system models, thus reducing the participant's inhibitions to communicate and to capture raising issues. The collaboration model elements are attached to related system model elements, which represent the context for the collaboration. Traces between collaboration model elements and system model elements are captured and maintained.

The *organizational model* describes the organizational structure of a software development project in terms of teams and participants. It depicts the relationships among participants, and between participants and the system and collaboration models they create and work with. By making this information explicit, participants can more easily contact relevant stakeholders and experts and resolve issues more quickly. By providing mechanisms to raise the awareness of discussions and issues to potential recipients, participants save time in anticipating issues and avoid an information overload.

To evaluate the hypothesis, we have developed and evolved Sysiphus [38, 128, 131, 21], a distributed tool suite that implements the RUSE model. Sysiphus provides a central repository for the RUSE model elements and online access for synchronous collaboration of project participants from distributed sites, as well as offline workspace support for asynchronous interaction. Sysiphus was used in several academic and industrial software development projects as well as for teaching software engineering.

The novelty of our approach is that system models, collaboration models, and the organizational model are given equal emphasis, live in a single, shared repository, and are represented within the same model. The unified representation enables us to provide the same set of traceability and awareness services for and across all three types of models. The RUSE model is easily extendable by new model elements while reusing existing services for traceability and awareness. The challenges of this approach are to capture sufficient information as a side effect of development while structuring this information for long-term use.

Chapter 2 presents the requirements for the rationale-based unified software engineering model and describes the RUSE meta-model that meets the requirements. Chapter 3 presents the RUSE model that integrates system, collaboration and organizational models. The model conforms to the meta-model described in chapter 2. Chapter 4 presents different views and usages of the RUSE model, including traceability and awareness support. Chapter 5 presents Sysiphus, the reference implementation of the RUSE model. Chapter 6 describes several applications of the RUSE model in academic and industrial context and presents the evaluation of the research. Chapter 7 presents related work and tools. In the last Chapter 8, we summarize our research and outline future directions.

CHAPTER 2

The RUSE Meta-Model

A *model* is an abstraction of phenomena of the real world. In this dissertation we present the RUSE model (see chapter 3) that integrates system models, collaborations models and organizational models to support distributed software development projects. In this chapter we present a *meta-model* for the rationale-based unified software engineering model. The meta-model addresses all requirements that the specializing RUSE model elements have in common.

First we introduce the RUSE model properties as a set of requirements. Than we define a meta-model that meets the requirements. In chapter 3 we define the RUSE model conforming to the meta-model, thus ensuring to meet all requirements. All RUSE model elements, like use cases or nonfunctional requirements from the system model, comments or issues from the collaboration model or participants and teams from the organizational model, extend the meta-model and use the provided meta-model mechanisms.

2.1 Requirements

We define the following requirements for all model elements of the RUSE model. By realizing the requirements in the meta-model, we ensure that all model elements meet the requirements.

Artifact integration

Many related artifacts are created and maintained during software development projects. We categorize the artifacts into system models, organizational models, and collaboration models. The system models describe the system under development and consist of models and documents. The organizational models describe the project organization in terms of teams and participants and their roles in the development project. In project management, the organizational models are

required to allocate activities and tasks to participants, which describe the development of the system models. Moreover, the organizational models are needed to identify experts and stakeholders. The project participants need to collaborate when developing the system models. The collaboration models include the development tasks that are needed to coordinate the project, informal discussion threads to rapidly exchange knowledge and to clarify arising problems, as well as formal collaboration models such as issue models to explore and evaluate the solution space of critical issues and to capture the rationale behind system relevant decisions.

Unfortunately, the related artifacts are created and maintained in different and separated models and tools, which make the expression of relations across artifacts difficult. Specialized tools are used to create the required system models and documents. Project management and organizational tools are disconnected from the system modeling tools and are usually not accessible for all participants. Collaboration and in particular communication occurs in separated channels and tools. The identification of system model relevant communication threads, their rationale, and their relevant participants is generally not supported.

The RUSE model integrates the described artifacts and is based on a meta-model. Thus, the meta-model must be capable to represent any of these artifacts and the artifact relations. When all artifacts are represented in one model, a single model repository can be used to maintain all artifacts and make them accessible to all project participants located on distributed sites.

Traceability

Software projects need to deal with change. Requirements change when the problem domain is better understood. The architecture is adapted when new technology is selected. To maintain consistency among documents and models, impacted elements need to be identified and updated [35]. Moreover, it is often necessary to identify the stakeholders responsible for these elements, as they may have implicit knowledge about how or whether the change should be realized. Tracing forward from requirements to impacted design, implementation, or test elements is called *posttraceability*. Tracing back from model elements to their human sources is called *pretraceability* [69]. The need for traceability among all development artifacts is clearly recognized in research [47, 48, 49] and traceability reference models have been developed [104].

But due to the variety of tools and models, a complete traceability across all system models, collaboration models and organizational models is not reached. Even if the dependency traces exist in one tool (for example in DOORS [121]), or in a separated file like a spread sheet (e.g. Microsoft Excel), there is no single view that shows all traceability relationships and the related artifacts. Changing or deleting artifacts that are referenced by external dependency traces lead to inconsistencies or even to dangling links, which need to be manually maintained. In

general, there are two possible approaches to support traceability across all artifacts. In the first approach, all used tools must be integrated and know the models of foreign tools, propagate changes across tools and models, and maintain traceability links. This solution is hardly reachable as long as the tools are produced by different and possible competing vendors. We follow the second approach, in which all artifacts are maintained in one model and tool. Integration between models or tools is not required. The dependency traces are captured within the model and are maintained automatically.

Nuseibeh et al. [91, 92] found in three case studies that not all inconsistencies can be eliminated in practice. They describe that sometimes the elimination of inconsistencies are more expensive and not applicable in projects under time pressure than living with inconsistent models and documents. The participants of the case studies marked inconsistent models to prevent their usage and the propagation of new subsequent inconsistencies. But Nuseibeh et al. also state that in some cases the elimination of inconsistencies are necessary. If inconsistencies must be eliminated or not must be decided by the relevant project participants. The most important point is the knowledge of existing inconsistencies, which is prerequisite for the decision if the inconsistencies must be eliminated or not.

We agree with the findings of Nuseibeh et al. and propose traceability links across all related system models, the organizational models, and the collaboration models to detect arising inconsistencies. When any element is the subject of change, related and depending elements can be identified by traversing over the traceability links. The participants can decide if related elements must be changed accordingly. If the participants are not able to decide due to insufficient knowledge, the traceability links between the system models and the organizational models facilitates the identification relevant experts or stakeholders. Moreover, related information and knowledge can be identified by tracing from the relevant system elements to their rationale and collaboration artifacts like comments, discussions, or related work items.

If inconsistencies between different models are identified, and the decision for elimination or not is made, the participants have to change the models accordingly. If they decided for elimination of the inconsistencies, they have two possibilities: They can change the models immediately or mark them with new tasks that describe the inconsistencies. The tasks can then be assigned to other participants and can be carried out later. If the inconsistencies should not be eliminated, the relevant models must be marked with a note. The traceability links facilitate the creation of tasks and notes to all related artifacts automatically.

Accountability and access control

When critical tasks such as the elimination of identified inconsistencies occur or question to specific model elements arise, the creators or the persons who modified these elements might have relevant implicit knowledge that is not docu-

mented. The identification of these experts is especially in distributed projects a problem and takes much longer than in collocated projects [54, 85, 53]. To speed up the identification of experts for the RUSE model elements, all creators, modifying authors, and the related dates and times must be identifiable for a given element.

Moreover, access control for the different kinds of model elements must be configurable for a project. Roles which are allowed to create, read, modify, or delete a kind of element such as a use case, a requirement or a class can be created and assigned to the project participants. Typical roles of a software development project include the requirements engineer, the architect, or the tester [19].

Awareness

A main issue in the collaborative activity in distributed development projects is *awareness*. Dourish and Bellotti [36] define awareness as "an understanding of the activities of others, which provides a context for [one's] own activity". Awareness includes knowing who else is working on the project, what they are doing, which artifacts they are or were modifying, and how their work may impact other work. The awareness problem is clearly recognized in research [72, 24, 52, 33, 118, 111].

The main prerequisite for supporting awareness in a project is to capture what the project participants are doing. Therefore, all elements of the RUSE model must be identifiable that were read, modified, or created by a specific project participant. Using this information in combination with the traceability links of the model elements and their accountability can be used to identify related participants. For example, when a tester modifies a test case that is related to a use case, which is related to a feature of the system, the requirements engineer of the feature can be related to the tester. The transitive relationship can be used to notify the tester when the feature is manipulated and provide the contact information of the requirements engineer for requesting additional information.

Consistency of multiple model views

Software development projects use different tools for different tasks. The tools range from modeling, testing or change tracking tools to word processing or presentation tools. For instance, a common practice for creating a requirements analysis document is to use a requirements tool like DOORS [121] in combination with a UML modeling tool like Rational Software Modeler [64] and a word processing tool like Microsoft Word [84]. The required model elements are created and maintained in their related modeling tools and are than added into documents and enriched with text to increase the understandability. It is common that model elements occur multiple times in one or even in many documents. For example, use cases or classes are presented in multiple use case or class diagrams. The diagrams are exported from the modeling tools into images and imported into Word.

In addition, the model elements are presented as structured text that describe the model elements in detail and also the text of the document sections usually refers to selected model elements.

We call the different occurrences *views* of a model element, which show the same model element in different contexts. For example, the exported model images are views of the model that is maintained in the related modeling tool and outside of Word. The documents get outdated as soon as the model gets changed. The images must be exported again, and all diagram occurrences in the documents must be replaced. Changing a name of a model element requires to update all name occurrences in the documents. The modeling tools have no support to capture all external views of their artifacts. Thus, the engineers changing a model in a modeling tool usually do not know which or how many views of the model must be updated. This problem gets worse when the views are geographically distributed and maintained by different participants. Maintaining view consistency between different tools with redundant information and different persistency mechanisms by manually reviewing all artifacts is costly for large development projects. Nuseibeh et al. [91, 92] describe a case study in which a document contained a diagram and a detailed textual specification of the diagram content. After changing the textual specification, the diagram was inconsistent with the text. Instead of updating the diagram, the project participants marked the diagram as outdated and referred to the textual specification.

Different views of the software engineering artifacts represented by the rationale-based unified software engineering model must always be consistent. This requires that each artifact is represented only once and the model has no redundancies. Formal models and informal text descriptions must be encapsulated in unique model elements with a unique identifier that can be referred to from any view. Thus, all views are always consistent as they represent the information of the same element. When changing an element, all referring views reflect the change immediately and the model views are kept consistent.

Support for fine-grained search and filter mechanisms

When all software engineering artifacts are represented in one model, the number of elements in the model become very huge for large projects. Fine-grained search and filter mechanisms are required to support the tasks of different project participants. For example, a project manager may be interested in all development tasks that are not finished and have a certain due date. An analyst would like to see all documents that contain views of a specific analysis class. A requirements engineer might be interested in all model elements that contain domain related words in either the description or in any attribute. Model quality reports might search for elements according to defined design guidelines and metrics [8, 9]. For example a quality report on use cases can find all use cases that are not related to

any actors. A quality report on classes might find all classes that have no instance in any sequence diagram or activity diagram [12].

The fine-grained search and filter mechanisms must be able to access and compare all relevant attributes of the different kinds of model elements from the RUSE model. Moreover, the mechanisms must be able to identify, traverse, and filter over all kinds of model element associations. At the same time, the mechanisms must not depend on the kinds of model elements. Otherwise the mechanisms must be updated or changed whenever a new kind of element is added or an existing is changed.

To realize such mechanisms, all model elements must have a uniform representation that is independent from the way the elements are used. The attributes and associations of a model element must be discoverable so that the generic search and filter mechanisms are applicable on all model elements. Therefore, the meta-model must provide a generic mechanism to store and retrieve any kind of data and to create associations to other model elements, without requiring any change. The uniform and generic representation of model elements is also required for the extendability of the model.

Extendability

This dissertation presents the rationale-based unified software engineering model that is adequate for executing a range of distributed development projects. However, the model may need to be changed, customized, tailored or extended, depending to a project needs and the used methodology. Adding new kinds of model elements and adding new kinds of associations to existing models must not require any changes in neither the meta-model, nor existing models.

Configuration management

It is widely recognized that software configuration management (SCM) is crucial for maintaining consistency among, and minimizing the risk and cost of changes to, *all* artifacts of a software development project [65, 66]. The fact that SCM is essential for the success of any and specially for distributed development projects is also reflected in the Capability Maturity Model Integration [114] that defines levels to assess the maturity of the software development process in organizations.

The currently available version control tools such as Subversion [123] or Rational ClearCase [59] are geared towards supporting textual artifacts such as source code. These systems are inadequate for many types of models such as requirements, use cases, feature models, architectures, class models, test cases or rationale. These artifacts have complex internal structures and semantic relationships. These SCM systems treat a software system as a set of files and directories and manage them in a line oriented way. This creates an *impedance mismatch* between

the simple, flat file-based data models in traditional SCM systems and software engineering models with complex internal structures and semantic dependencies [90]. Some specialized repository-based tools such as DOORS [121] for requirements provide SCM functionality such as baseline, track, or control changes. But not all kinds of artifacts such as collaboration artifacts can be integrated in these tools. Integrating different SCM tools for different kind of artifacts leads also to numerous problems, as different tools do not share the same version space.

Thus, the meta model must include a configuration management model that, according to Dart [31, 32] and Estublier et al. [41], supports the SCM categories *versioning*, *system models and selection* and *workspace control*. Each model element conforming the meta model must be part of the configuration management model that supports a version history, version selection, change tracking, the identification of differences between versions, workspaces, conflict detection and resolution, baselining, branching and merging.

Configurable labeling and categorization

Different organizations and even different projects of the same organization with different clients have different terminologies for the same software engineering artifacts. The names of the software engineering artifacts used in a development project must conform to the terminology of the project. For instance, a global nonfunctional requirement that is not directly related to the system under development, but constrains the way of development is sometimes called *pseudo requirement* and sometimes called *constraint*. We define the name of the artifact class as *label*. The label of each model element class must be configurable within a project at runtime.

Sometimes, specially in large projects, a custom categorization mechanism is needed to maintain clarity, by clustering a large amount of elements. Just the label of categorized elements differ, while the element structure - attributes and associations - remain the same. For example, a common practice is to categorize the nonfunctional requirements into usability, reliability, performance, and supportability, [50, 67]. The label of the elements should depend on the category and should not be nonfunctional requirement. A nonfunctional requirement of the usability category should be labeled as *usability nonfunctional requirement*. The model elements must be filterable by their category so that all *usability nonfunctional requirement* or *constraints* can be listed. In addition, to visually separate different categories, a project must allow to configure custom icons for a each model element categorization. The categories of model element classes must be dynamically configurable for a project.

2.2 The Meta-Model

Before defining the meta-model in detail (see sections 2.2.1 and 2.2.2), we provide a conceptual overview.

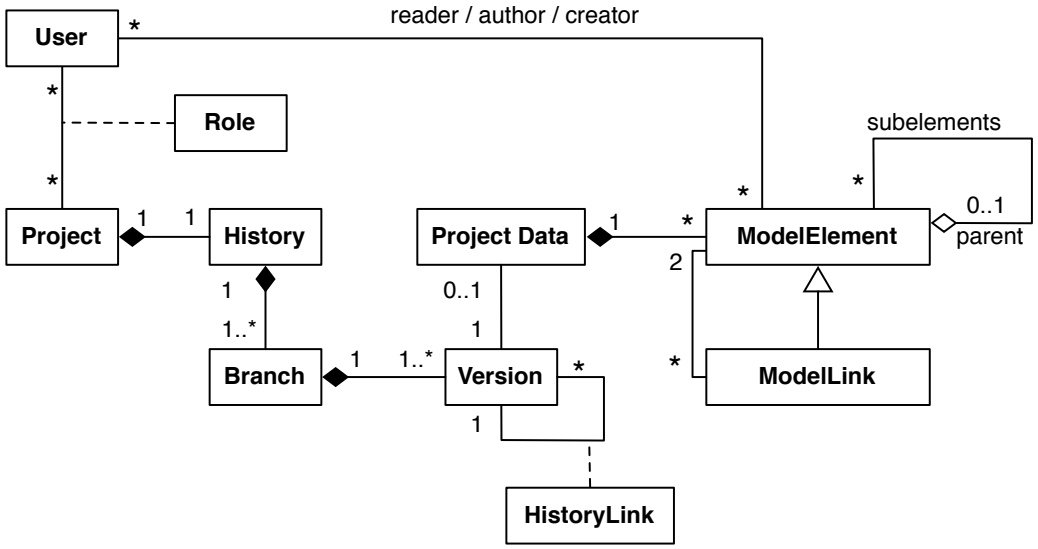


Figure 2.1: Meta-model overview (UML class diagram)

We start the description with the central class Project, which represents a software development project, containing all project related entities. It provides access to the project data, the history containing different versions and their changes, as well as to the users that access the project.

Instances of the class User represent persons or external entities, accessing the project. A User may access many Projects, defined by the Role association.

The Role association class relates many Users with many Projects. The Role defines the user's role in a project. Typical roles are *Project Manager*, *Requirements Engineer*, *Analyst*, or *Architect* [19].

The History class represents the history of the Project. It provides operations for creating revisions, branches, tags and for accessing specific versions, differences and history information.

The Branch class represents a branch of concurrent development in the version space and is composed of all versions that are available in the branch.

Versions represent the data of the Project at a specific state. A Version can have many successor Versions that are associated with the History Link class. The difference between two associated Versions is called *delta*. The Versions and the History Links represent the *version graph* of a Project. The data of the Project at a specific

Version can either be represented explicitly by a Project Data instance or implicitly by its position in the version graph and the appropriate deltas, represented by the Change Packages (see section 2.2.2).

The class Project Data is composed of Model Elements and Model Links, which associates two Model Elements. The Model Elements and Model Links build a graph structure. The Project Data class provides operations for accessing, searching and filtering Model Elements and for traversing over the Model Links.

The class Model Element is the most abstract and generic modeling class that represents any concept of the software engineering domain. A Model Element can have many child Model Elements and can be linked by Model Link classes to many other Model Elements. The Model Element class provides generic methods to store and retrieve arbitrary data.

The Model Link class is an association class that links two related Model Element instances. The Model Link class extends the Model Element class and inherits to all its properties to represent a software engineering concept.

The History Link class defines the successor Versions for a given Version and represents the edges in the version graph. The History Link has the two subclasses Revision Link and Variant Link that are defined in 2.2.2. Revision Links represent changes between two Versions within the same branch, while the Variant Link connects two Versions from different branches.

In the following sections, we define the RUSE meta-model in detail. Section 2.2.1 defines the project data model that provides the meta-model entities for creating software engineering artifacts. Section 2.2.2 discusses different configuration management approaches and leads to the version object model.

2.2.1 The project data model

This sections defines the project data model of the meta-model. The project data model provides a generic mechanism to create software engineering artifacts that meets the requirements defined in section 2.1. The main entities of the project data model are the classes Project Data, Model Element, and Model Link. We also add and describe the associated classes Project and User, as they provide a good entry point to the project data model. We describe all entities in detail by natural text, UML [93] and the Object Constraint Language (OCL) [94]. Figure 2.2 shows a UML class diagram of the model.

Project

The **Project** class represents a software development projects and is the base class of the meta-model for creating and integrating various models. From the Project class, all project related entities like the History, the Project Datas, the Model Elements, and generic services for exploring, searching, filtering, and analyzing the meta-model are accessible. A Project is associated by the Role association class with many Users and is responsible for a role-based access control. Each Project

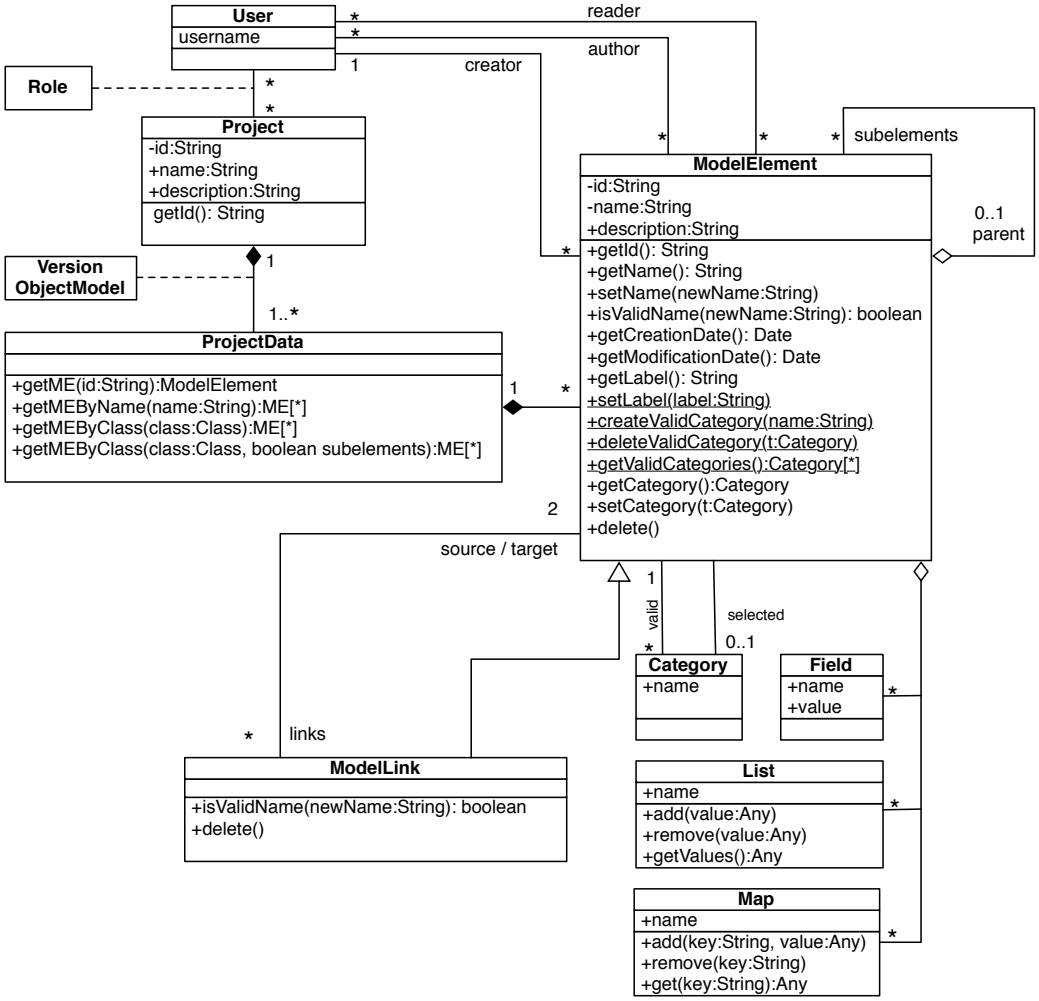


Figure 2.2: The project data model

The project data model of the RUSE meta-model and its associated classes Project and User as an UML class diagram. Note that the association between the Project and Project Data is an association class that represents the version object model of the meta-model. The association results from several transitive association of the version object model described in section 2.2.2.

instance has a name, a description, and a global unique id, which is not modifiable. During instantiation, the id and the initial version object model, described in section 2.2.2, are created. By traversing associations, a new Project can access one History, one Branch, the first Version and the initial empty Project Data.

Listing 2.1: OCL constraints of the class Project

```
context Project inv:
  Project.allInstances()
    ->forAll(p1, p2 | p1<>p2 implies p1.id<>p2.id)

  self.history.branch->size() >= 1
  self.history.branch.version->size() >= 1
  self.history.branch.version.projectdata->size() >= 1
```

User

The **User** class represents persons or external entities which have access right on a Project. Many Users participate in different roles in many Projects. Depending on their role, the users have different access rights to the Project and its contained entities like Model Elements. Depending to the project needs, different roles and role-based access policies must be dynamically defined. The User instances are identified by a unique username. A User can create, modify and access the Model Elements of the Project. All these interactions are captured by an association, containing the date and time of the interaction. These associations enable the identification of all Model Elements accessed by a User, and the identification of all Users that interacted with a specific Model Element. This information is essential for supporting awareness and identifying experts in a Project.

Listing 2.2: OCL constraints of the class User

```
context User inv:
  User.allInstances()
    ->forAll(u1,u2|u1<>u2 implies u1.username<>u2.username)
```

Project Data

The **Project Data** class represents the complete data of a project in a specific version. It is composed of Model Elements and Model Links, which relates two elements, thus building a graph structure. A Project has at least one Project Data that that holds the latest version of the data. The Project Data class provides operations for accessing, searching and filtering all containing Model Elements. Queries for searching and filtering are based on the discoverable Field, List, and Map properties, the Model Link graph structure, the hierarchical subelement structure, as well as on the class, label, and category information of a model subclass. The Model Element properties used for queries are defined in detail below.

The field, list, and map values, the subelement and link structures, as well as the class names of the model classes and links can be used in queries to narrow

down the search space. As system, collaboration, and organizational models are part of the same meta-model graph, these services can be used to bridge the gap between modeling, collaboration, and awareness.

Listing 2.3: OCL constraints of the class ProjectData

```
context ProjectData inv:
  ProjectData.allInstances()->size() >= 1
```

Model Element

The **Model Element** class is the super class of all model classes and provides the main meta-model operations. Each Model Element has a unique id in the scope of its Project Data. Model Elements may have the same id in different Project Data instances, as different Project Datas represent the same data in different versions. Each Model Element has a name and a description attribute. The operation +isValidName(name:String):boolean defines if the provided String parameter is a valid new name for specific Model Element instance. In the specification of the Model Element, a valid name is not empty and no other instance of the the class on which the operation is invoked, has the name. For instance, if +isValidName(newName:String):boolean is invoked on the class Use Case, a model subclass of Model Element, the name is valid if no other Use Case exist with the *newName*. The operation +setName(newName:String) tests if the new name is valid before changing the name. The operation +isValidName(newName:String):boolean should be overloaded by subclasses to redefine the valid name specification if needed. For instance, the name of a class must only be unique within its package and not among all existing classes (see listing 2.4).

The operation +getLabel():String returns a default end-user readable artifact name for the Model Element. Subclasses should overwrite the method and return an appropriate name. For instance, the class Use Case returns "Use Case" and the class NFR returns "Nonfunctional Requirement". To conform to the terminology used in a project, the label can be changed on runtime by the static class operation +setLabel(label:String). The operation changes the label for all instances of the same class. For instance, the label of the class NFR can be changed to "Constraint". Moreover, when overwriting the operation +getLabel():String, instances of a subclass of Model Element can return a label that is based and reflects the state of the instance. For instance, Brügge and Dutoit [19, p. 498] define the class Action Item with subject, description, owner, deadline, and status attributes. The owner is the person responsible for completing the Action Item. The status of an Action Item can be todo, notDoable, inProgress, or done. When realizing the class Action Item within this meta-model, the class Action Item can overwrite the +getLabel():String operation and return a label that is based on the state. Thus, Action Item instances with the state todo, can visually be labeled as "ToDos", while other instances with a deadline and an owner are labeled as "Action Items".

The Model Element class may be associated with any number of Fields, Lists, or Maps. In the context of a Model Element instance, each Field, List, and Map instance has a unique name, which is used to access it. Adding a field, list, or map replaces an exiting field, list or map, if the name of the new one already exists. A field is used to store any named value in a Model Element instance. A list is used to store a set of values referring to the same name. Within a map, any key-value pair can be stored. Fields, Lists, and Maps are used by subclasses to store any kind of data. Subclasses should hide the usage of fields, lists and maps by encapsulating the usage in operations. For instance, the subclass Class, from the system model, provides the operations +isAbstract():boolean and +setAbstract(abstract:boolean) to identify and define if the Class instance is abstract or not. The operations use the provided meta-model mechanism to access and store the boolean value within an associated Field instance, named "abstract". The encapsulation of Fields, Lists, and Maps enables subclasses to provide a clean API for the related modeling domain, while using the generic mechanism of the Model Element class takes cares about persistency and provides generic operations to explore the meta-model.

Listing 2.4: OCL constraints of the class ModelElement

```
context ModelElement inv:
  ModelElement.allInstances()->forAll(e1, e2 |
        (e1<>e2 implies e1.id<>e2.id)
      or
        (e1<>e2 and e1.id=e2.id
          implies e1.projectdata<>e2.projectdata
        )
      )

  field->forAll(f1, f2 | f1<>f2 implies f1.name<>f2.name)
  list->forAll(l1, l2 | l1<>l2 implies l1.name<>l2.name)
  map->forAll(m1, m2 | m1<>m2 implies m1.name<>m2.name)

context ModelElement::setName(newName:String)
  pre:  isValidName(newName)
  post: !isValidName(newName)

context ModelElement::isValidName(newName:String):boolean
  pre: self.name <> newName
  post: result = self.getType().allInstances()
                 ->forAll(e1 | e1.name <> newName)
                 and newName->size()>0

context ModelElement::delete()
  post: links->size()=0
  post: subelements->size()=0
```

To support a hierarchical structure between Model Elements, each element may have an arbitrary number of subelements associated, and each element has at most one associated parent Model Element. The associated subelements are ordered and the hierarchical structure and order can be dynamically changed. The hierarchical structure of the meta-model does not implicate a defined meaning. A model of the meta-model can use the hierarchical structure to realize any required relationship. The only implicit meaning of the hierarchical subelement relationship comes with the +delete() operation of the Model Element.

The +delete() operation deletes a Model Element instance from it's Project Data, and deletes recursively all associated subelements of the Model Element. Therefore, the hierarchical subelement association mechanism fits best for a hierarchical composition relationship, in which the subelements are bound to the existence of the parent Model Element. Example relationships are the relations between UML classes and their attributes and operations. A class Class would represent the parent of the subelement classes Attribute and Operation. Each Attribute and Operation instance has exactly one parent element Class, while a Class may have many subelement Attribute and Operation instances. Deleting a Class instance will also delete all it's Attribute and Operation instances.

Model Link

The **Model Link** class is used to associate any related Model Elements. The Model Link connects two Model Elements, realizing a graph structure in which the Model Elements are nodes and the Model Links are edges. A Model Element instance may be associated with any number of Model Link instances, while each Model Link has exactly two associated Model Elements, called source and a target. The Model Link extends the Model Element and therefore, it is also a full-fledged Model Element, which may be used to store any fields, lists, or maps. For example, a Model Link is used to realize UML associations between UML classes. The data of the association like the multiplicities or roles are by the field mechanisms inherited from the Model Element class. Moreover, a Model Link instance may have any subelements, as well as links to other Model Element instances.

When a Model Element instance is deleted by it's +delete() operation, all it's associated Model Link instances get deleted too, while the opponents of the Model Links are not affected. The Model Link refines the +delete() operation to ensure that source and target Model Elements remove their reference to the link (see listing refocl:ModelLink). As the +delete() operation of the Model Element ensures that all it's links are deleted too, the +delete() operation invokes the +delete() operation of all its Model Links. In addition, the Model Link refines the +isValidName(newName:String):boolean operation. For a Model Link instance, all names are valid that differs from current name.

Listing 2.5: OCL constraints of the class ModelLink

```
context ModelLink::delete()
```

```
pre:    source.links->exists(l:ModelLink | link=self)
pre:    target.links->exists(l:ModelLink | link=self)
post: @pre.source.links->forAll(link:ModelLink | l<>self)
post: @pre.target.links->forAll(l:ModelLink | l<>self)

context ModelLink::isValidName(newName:String):boolean
pre:    self.name <> newName
post: result = true
```

To separate between link classes and to ensure type consistency, a model should realize links between elements by extending the Model Link with a new link class. While the meta-model provides a generic mechanism for storage and link traversal, the model subclasses are used for type checking and for providing the specialized model dependent behavior. The arising link taxonomy can be used in search queries, and in filtered link traversals. Figure 2.3 shows an example model, extending the meta-model. The classes Use Case and Actor represent the UML use case and actor and extend the Model Element class. A use case can be initiated by one actor, which is represented by the specialized link class InitiatingActorUseCaseLink. The specialized link inherits the ability to link two elements and validates that the linked element classes are a Use Case and an Actor.

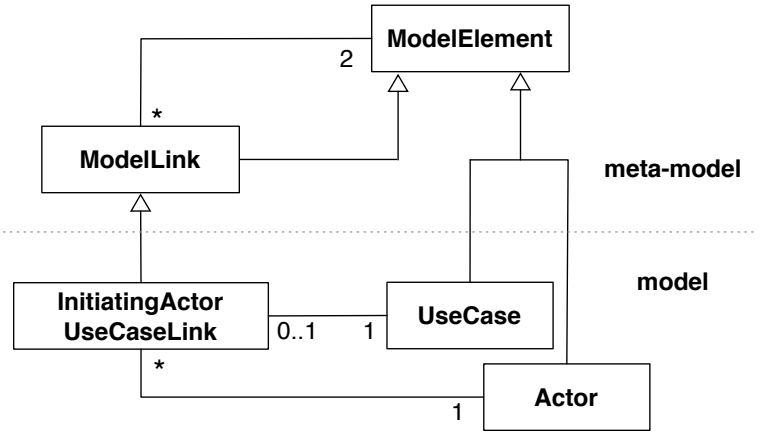


Figure 2.3: Example extension of the meta-model

2.2.2 The configuration management model

Software configuration management (SCM) is a well accepted technique for managing change in software development projects. It is part of the Capability Maturity Model Integration [114] that defines levels to assess the maturity of the software development process in organizations. SCM systems follow different

approaches concerning *delta representation* and *granularity*, *version granularity*, the *version object model*, and the handling of *diffing and merging*, and *conflict detection and resolution* to realize configuration management.

This section discusses and evaluates various SCM techniques to be used for the RUSE meta-model. We follow Conradi and Westfechtel's SCM framework the *uniform version model*, presented in [26, 27]. The framework provides a common terminology and classification with which the available alternative approaches for the SCM design can be uniformly expressed and compared. The terminology is widely accepted and cited within the SCM community. We extended the categorization of the original framework by adding the aspect of delta representation since it is highly relevant for fine-grained configuration management of complex data structures, like the RUSE Model. After evaluating different approaches, we describe the resulting version object model of the RUSE meta-model.

Delta representation

Deltas are the differences between a configuration item in two different versions. Deltas can be represented using one of two basic approaches, *state-based deltas* or *operation-based deltas*. The differences between the two approaches are very subtle in many SCM systems but are highly relevant.

State-based Deltas In the state-based approach, only the state representations of different versions are stored, possibly using compression or sharing of common parts. Deltas are reconstructed using a differencing algorithm that compares the different state representations.

Operation-based Deltas In the operation-based approach, changes are described by using the original sequence of operations that caused the changes. The operations are created by the editor application that is used to change data.

With state-based deltas, the semantic context of the original operations that caused the change has to be recalculated with the deltas. This approach is expensive and in some cases, it does not work at all [77]. For example, it can be impossible to unambiguously recalculate the original sequence of change operations, when the state changes of one operation are partially or completely masked by those of a later operation. This problem is of particular importance for the RUSE model, where state of Model Elements is internally represented by the RUSE meta-model. A single change operation actually results in a non-atomic series of transformations to that state.

For such multi-level data models that have a complex internal structure, the state data is often stored in a structured way, e.g. using XML, to preserve more contextual information to assist in the reconstruction of the original semantic context of the deltas. This approach is used in similar systems described in [96], [82],

[99] and [75, 76, 77]. However, even this improved approach can not resolve all ambiguities and remains complicated.

Storing the original editor operations automatically captures the original semantic context of the changes. When using the operation-based approach, deltas can easily be recorded on the model level. Several other research efforts have successfully employed the operation-based approach in environments, similar to the RUSE model [105, 90, 95].

A drawback of the operations-based approach is that the operations depend on the editors used, resulting in coupling the editor tools with the SCM engine. However, on the one hand this can be resolved by defining a standardized language to express changes in the means of operations, and on the other hand, it is not very common to use different editors to manipulate the same software engineering artifacts within the same project.

The editors have to support the recording of operations, which is usually not provided in systems with a simple interface to the SCM system, or in systems that have to support arbitrary editors. This is probably the main reason why operation-based systems are not in widespread use today.

The benefits of the operation-based approach clearly outweigh its drawbacks. Therefore, we select an operation-based delta representation. It is important to note that operations need to fulfill two requirements in order to be usable in operation-based deltas [78]):

- The operations have to be deterministically replayable in order to be used in forward deltas.
- The operations have to be reversible in order to be used in backward deltas.

Delta granularity

Another important question regarding an SCM system is the granularity used to describe changes. This is called the *delta granularity*. When changing software engineering models that are part of the RUSE model and conform to the RUSE meta-model, changes take place on three different semantic levels of granularity that are described below. Figure 2.4 illustrates the levels of change on an example.

Logical level These changes are sets of logically coherent work as seen by the user, e.g. “I updated the use cases, their analysis and the glossary according to today’s client review”.

Model Level These changes are atomic changes as far as a specific modeling domain is concerned, e.g. “set a new initiating actor for a use case”. They correspond to domain specific operations on the Model Elements of the RUSE model and are usually comprised of several changes on the meta-model level.

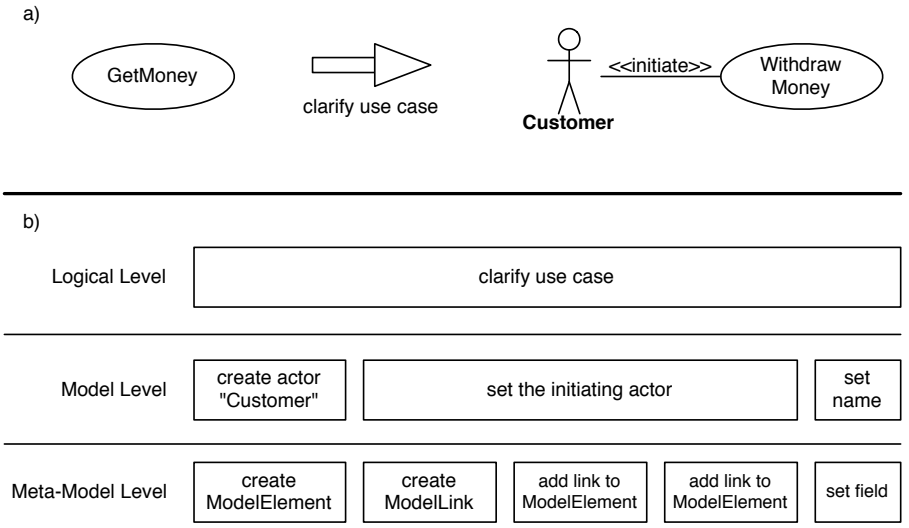


Figure 2.4: Delta granularity: Three semantic levels of change

Illustration of the delta change granularity that involves three semantic levels. Figure 2.4 a) shows an example of change. The use case "GetMoney" describes the interaction of a bank customer with an ATM machine. In our example, a requirements engineer clarifies the use case, by creating a new actor "Customer", setting the customer as initiating actor for the use case and changing the name of the use case to "WithdrawMoney". Figure 2.4 b) shows the delta granularity of change on the three semantic levels. On the logical level of the requirements engineer, one change occurs: the clarification of the use case. The clarification of the use case involves more changes on the model level. The new actor "Customer" is created, the "Customer" is set as initiating actor of the use case, and the use case is renamed. All changes of the model level are transformed to changes on the meta-model level. Creating a new actor transforms to the creation of a new ModelElement. Setting the initiating actor creates a new ModelLink and adds the link two the ModelElement instances representing the use case and the actor. Changing the name sets a field value on the meta-model level.

Meta-model Level These are the changes as seen by the meta-model. They change attribute values of single Model Elements. Users of the system are usually not aware of and do not understand this level of change.

The SCM approach needs to be able to describe and track changes on all three levels of granularity. Fine-grained change tracking can easily be achieved on the meta-model level since changes can be described with the granularity of changes to single attributes of a Model Element. Describing the meta-model changes has the additional benefit of being independent of the underlying model level. We use the class Meta Model Operation to describe changes on a Model Element.

Unfortunately, this alone is not sufficient since it does not capture enough context. A meta-model change on its own will not be meaningful to a user of the system since he will be working on the semantic level of the model and generally

not be aware of the mechanics of the meta-model. Reversing the original model level changes from a series of meta-model changes is a difficult task and nearly impossible, since it would require the SCM engine to have detailed knowledge of every domain model on the model level. Furthermore, operations on the logical model level often do not have an injective mapping to the meta-model, making an unambiguous reconstruction impossible. As an obvious example, the removal of an element in one part of the meta-model graph and the addition of a similar element in another part could be the result of a move operation, as well as the result of a delete and add operation.

Therefore, we use the Model Operation class to capture additional information and to preserve the full semantic context of the model level. The model level can add additional type specific integrity constraints that need to be considered during conflict detection and resolution. For example, a use case can have only one initiating actor or a class can not generalize itself. These constraints can only be checked with additional domain knowledge from the model that is not known by the meta-model.

An SCM system can automatically track and describe changes on the meta-model and model level, but not on the logical level. Therefore, the SCM approach must provide a mechanism for manually grouping and describing logical changes by the user. Our approach provides Change Packages with log messages to achieve this on the logical level of granularity.

Figure 2.5 shows the classes Meta Model Operation, Model Operation, and Change Package with their associations to the classes from the project data model. The classes are described in detail below.

Abstract Operation

The Abstract Operation class is the abstract super class of the Meta Model Operation and Model Operation classes. The Abstract Operation class and its subclasses build a composite pattern [46], in which the Abstract Operation describes an abstract change. The operation +reverse():AbstractOperation creates a new Abstract Operation, which describes reverse change of the origin change.

Meta Model Operation

Meta Model Operations describe changes on the semantic level of the meta-model. They describe a single change that affects and changes exactly one Model Element. A Meta Model Operation support reversibility as described above. The operation +reverse():MetaModelOperation creates a new Meta Model Operation that can be used to revert the Model Element change. The Meta Model Operation class extends the abstract class Abstract Operation and represents the leaf of the composite pattern. The operation class is abstract, as the concrete changes of a Model Element are described by several Meta Model Operation subclasses. The Meta Model Operation taxonomy is shown in figure 2.6.

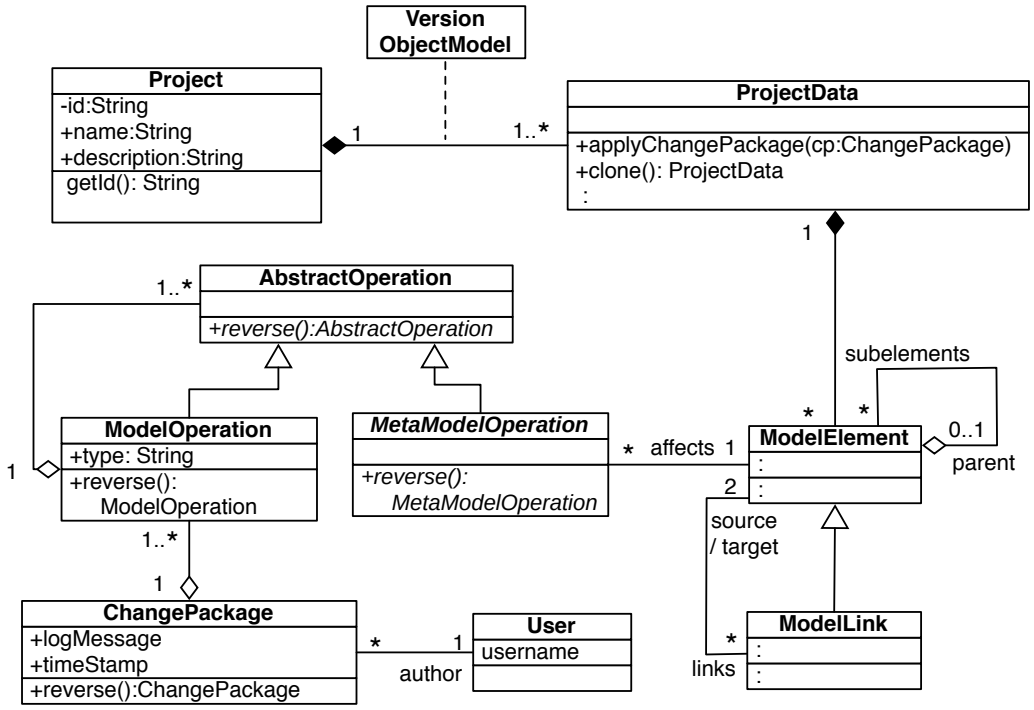


Figure 2.5: Representation of deltas on three semantic change levels.

The UML diagram shows the classes and associations that are used to represent changes of the project data model. The Meta Model Operation describes changes on the meta-model level, the Model Operation describes changes on the model level, and the Change Package describes the logical change level from the user. Note that the association between Project and Project Data is an association class that represents the version object model described below.

The CreateOperation is used to describe the creation of a new Model Element. It's +reverse():MetaModelOperation creates and returns a DeleteOperation, which is used to describe the deletion of a Model Element. Reversing a DeleteOperation returns a CreateOperation of the affected Model Element.

The FieldOperation is used to describe Field changes of a Model Element. The type attribute indicates whether a field gets added or removed. The name of the field is stored in the fieldName attribute, while the new and old field value is stored in the newValue and oldValue attributes. The +reverse():MetaModelOperation returns a FieldOperation, which changes the Field of the Model Element into it's previous state.

The ListOperation is used to describe List changes of a Model Element. The type attribute indicates whether a list entry gets added or removed. The name of the list is stored in the listName attribute, while the list value is stored in the listValue attribute. The +reverse():MetaModelOperation returns a ListOperation, which has the opposite type attribute value than the origin operation. Thus, the reversed

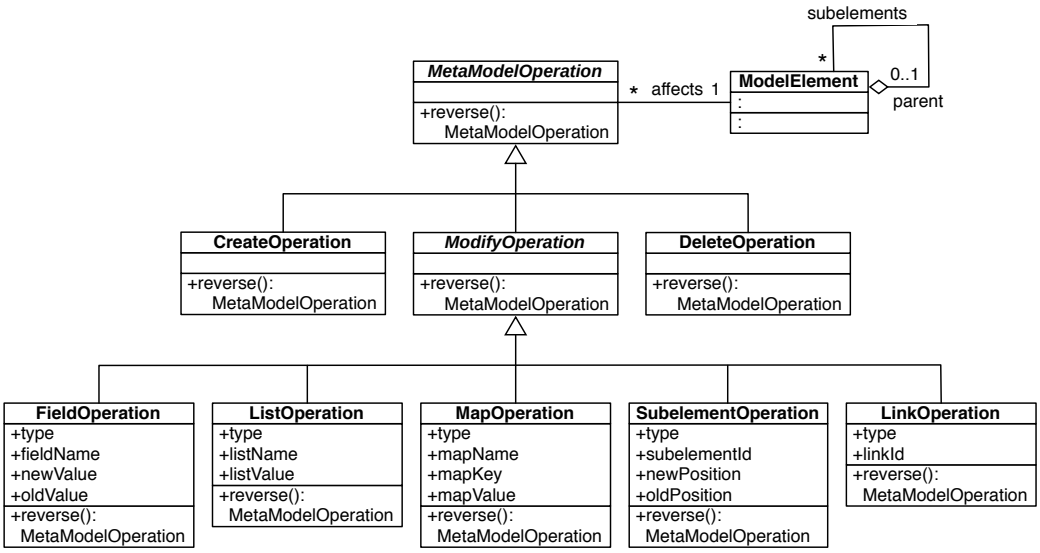


Figure 2.6: Taxonomy of Meta Model Operations

The UML class diagram shows the subclass taxonomy of the Meta Model Operation class, which reflects the possible changes of a Model Element instance, described in section 2.2.1.

ListOperation removes the list value entry when the origin operation added the value and adds the value when the origin removed it.

The MapOperation is used to describe Map changes of a Model Element. The type attribute indicates whether a map entry gets added or removed. The MapOperation has the attributes mapName, mapKey and mapValue, to hold the name of the modified map, the key and the value. The +reverse():MetaModelOperation returns a MapOperation, which has the opposite type attribute value than the origin operation. Thus, the reversed MapOperation removes the map entry when the origin operation added the entry and vice versa.

The SubelementOperation is used to describe changes on the hierarchical structure of a Model Element. The operation describes the changes on the parent Model Element, while the id of the subelement is hold in the subelementId attribute. The type attribute defines if a subelement is added, removed, or if the position of a subelement changed. The old and new position are captured with the oldPosition and newPosition attributes. The +reverse():MetaModelOperation creates and returns a SubelementOperation that describes the reversed change.

The LinkOperation describes changes of the link structure between Model Elements and Model Links. It type attribute indicates that a link is added to, or removed from the affected Model Element. The id of the related link is hold in the linkId attribute value. The +reverse():MetaModelOperation creates and returns a LinkOperation with the opposite type attribute value than the origin operation.

Model Operation

The Model Operation describes changes on the semantic level of the model layer. For example, it can describe the creation of an attribute on a class, or setting an exiting actor as the initiating actor on a use case. It extends the class Abstract Operation and adds an additional type attribute to capture the semantic context of the model layer operation that caused the changes. The Model Operation class is the aggregate of the composite pattern and is composed of one or more Abstract Operations, which are either Model Operations or Meta Model Operations. The containing Meta Model Operations describe the actual changes on the Model Elements at meta-model level, that are needed to perform the model change. For example, adding an attribute to a class is described in one Model Operation, which contains a Meta Model Operation to create the new attribute and one Meta Model Operation to add the new attribute as a subelement on the class. The Model Operation contains other Model Operations, when the model change requires additional changes on the model level. Assuming the model transformation *Extract Class* (see figure 2.7) is realized in the RUSE model. It would create one Model Operation "Extract Class" that is composed of other Model Operations like "Delete Attribute", "Create Class", "Add Attribute", and "Create Association".

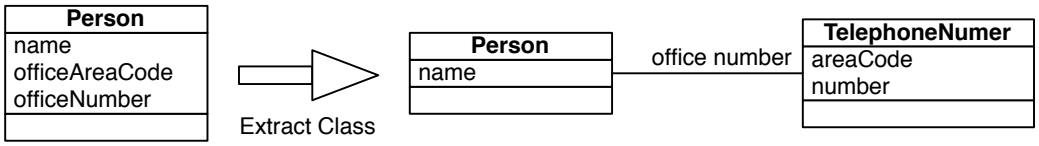


Figure 2.7: Example model transformation "Extract Class"

The model transformation *Extract Class* [43, p. 149] removes the attributes officeAreaCode and officeNumber from the class Person and creates a new class TelephonNumber with the attributes areaCode and number. Than, the new class is associated with the class Person.

Change Package

A Change Package represents a logically coherent set of work as seen by the user. The user has to provide a textual message that is set to the +logMessage attribute and describes the work. The Change Package is associated with its author and provides information about the time of the change, captured in the +timeStamp attribute. It is composed of one or more Model Operations, which describe the concrete changes on the model. Reversibility is supported by the operation +reverse():ChangePackage, which creates a new Change Package that is composed of all reversed Model Operations of the origin Change Package.

Model Element and Model Link

The classes Model Element and Model Link are the main classes of the meta-model.

They are described in detail in section 2.2.1. We add a new association between the Model Element and the Meta Model Operation class, which identifies all changes of a Model Element in a specific Project Data version.

Project Data

The Project Data class represents the versioned part of project data model. It is composed of Model Elements and Model Links. In order to facilitate the versioning mechanisms, we add the operations +clone:ProjectData and +applyChangePackage(cp:ChangePackage). The operation +clone:ProjectData creates a new Project Data instance, which is an exact copy of the instance on which the operation was invoked. All Model Elements and their states are also cloned and composed into the new Project Data instance. The operation applyChangePackage(cp:ChangePackage) applies the changes, described by the Change Package cp on the Model Elements of the Project Data.

Version granularity

The version granularity describes how product and version spaces are combined. Conradi and Westfechtel [26] present three types. One of the main issues here is how to represent configurations.

Component versioning In *component versioning*, each configuration item has its own separate version space. Since the version spaces of different configuration items are not directly related, there is no intrinsic method of providing consistent configurations. This must be added on top of the version model. An example of a system that uses component versioning is the Revision Control System (RCS) [45].

Total versioning *Total versioning* is an extension of component versioning in which composite items are versioned as well. Since configurations are composite items, they can be versioned directly in this approach. However, configurations are specified explicitly, thus, they add some additional complexity and effort to the management of configurations. An example of a system that follows this approach is Rational ClearCase [59].

Product versioning In *product versioning*, all configuration items share a common *uniform, global version space*. Thus configurations are represented implicitly and are always consistent, assuming no inconsistent versions are checked into the repository. This approach is less flexible than the other approaches but considerably simplifies the management of consistent configurations. Product versioning has become more and more popular with systems such as Subversion [123].

Component versioning lacks intrinsic support for managing consistent configurations. This is however highly desirable. The Model Element instances in a

Project Data are highly interdependent through semantic relationships and we are interested in managing sets of artifacts that consistently model an entire software project.

Total versioning is not a big improvement in this respect since it still requires explicit management of consistent configurations. The Model Elements managed by the system should completely and unambiguously describe exactly one system under development. Therefore, there is usually only one valid configuration at any point in time. Thus, the flexibility of being able to explicitly manage configurations is not needed and actually becomes a disadvantage by unnecessarily increasing the complexity of the system.

Offloading this task of managing the configurations to the user would make the system usage very difficult and error prone. Logic for supporting the user in this task could be built into the SCM engine, but that would be very time consuming and the process would still remain tedious. Many problems of this approach are described in [90]. Furthermore, this would imply that the SCM engine needs to be able to handle the structure and semantic integrity constraints of configuration items and configurations. However the SCM engine should be kept as independent of the internal structure of the data model as possible.

Product versioning lacks any modularity of the version space since it only has one uniform global version space for all configuration items. This non-modularity has its advantages and disadvantages. The main advantage is that version spaces of different configuration items are naturally related, alleviating the need to find combinations that produce valid configurations. Product versioning thus automatically provides consistent configurations without the need for explicit management of configurations or the SCM engine having to know about the exact nature of the data model. It can guarantee consistent configurations by always versioning the complete state of a project. Furthermore, product versioning is a natural match with Change Packages since both approaches handle changes that are spanning over several configuration items, the Model Elements, as a coherent entity.

One disadvantage of product versioning is that variants need to be global, too [27]. In our case this is not an important concern since varying single data Model Elements is seldom required as there is usually only one valid configuration at any point in time. In the few cases where this is required, such variants can easily be provided by branching. Another drawback of product versioning is that no unique version or history exists per configuration item. For the former, the last product version that changed the item can be used. For the latter, filters can easily be used that select only the changes for the specified elements when extracting history information.

Summarizing the analysis in this section, product versioning provides the version granularity most suitable for configuration management of the RUSE Model. As we have seen, it has several benefits and all its drawbacks can be overcome. We therefore use product versioning in our approach.

Version object model

The version model resulting from the analysis above is shown in the UML class diagram in figure 2.8. The version model classes are shown with their associations to classes from the project data model, introduced in section 2.2.1.

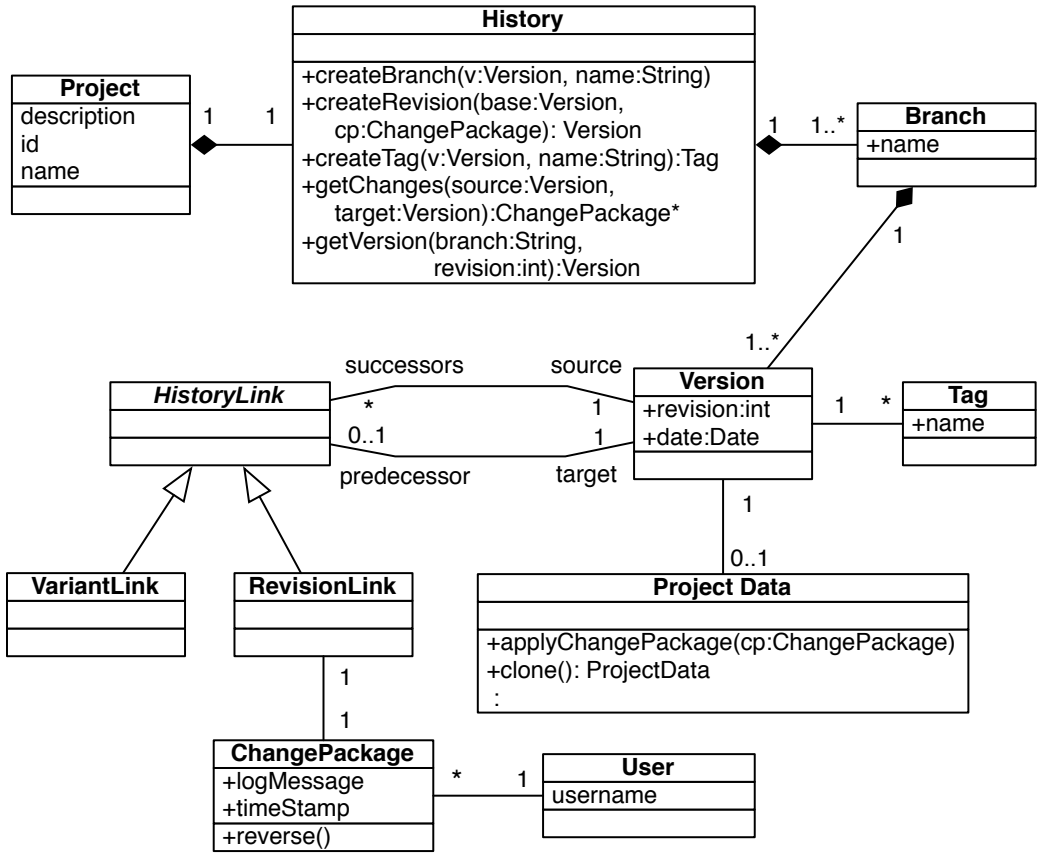


Figure 2.8: Version object model

The UML class diagram shows the version object model of the RUSE meta-model. The version space is represented by a graph structure consisting of Branches and Versions as nodes, while the Revision Links and Variant Links represent edges for the revision and variant relationships. The Revision Links are associated with the Change Packages, describing the changes between two Versions.

History

The History class represents the history of a Project. It provides operations for creating revisions, branches and tags and for accessing specific versions, differences and history information. A History has at least one Branch, which contains the main trunk of development. All Branch instances of a History must have different names.

The operation +createRevision(base:Version, cp:ChangePackage): Version creates a new Version from the base Version parameter. The base Version and the new Version get connected by a Revision Link, which gets associated with the Change Package parameter cp . The Revision Link indicates the successor Version for the base Version. The new Version is added to the Branch of the base Version. Both Versions might have an associated Project Data instance, representing the data model in two different versions. To create the new Project Data instance of the target Version, the +clone(): ProjectData operation is used to create an identical Project Data instance of the base Version, followed by the applyChangePackage(cp:ChangePackage) operation, changing the Project Data according to the Change Package associated with the Revision Link.

The operation createBranch(v:Version, name:String) creates a new Branch for concurrent development, which starts with the Version v. The name parameter must be unique among all existing Branches. A new Version is created in the new Branch and is connected by a Variant Link instance to the source Version v.

The operation +createTag(v:Version, name:String):Tag creates and returns a new Tag object with the name parameter. The name must be unique among all Tags in the History.

The operation +getChanges(source:Version, target:Version):ChangePackage* returns all Change Packages that are associated to the Revision Links, when traversing over the History Links from the source to the target Version. The result represents the delta between the two Versions. Regarding the example object diagram shown in figure 2.9, the +getChanges(...) operation from Version v1 to Version v2 returns the Change Packages cp1 and cp2. The changes between the Versions v1 and b2 is represented by the Change Package cp3.

The operation +getVersion(branch:String, revision:int):Version returns the Version instance that is identified by the branch and revision parameters, if existent.

Listing 2.6: OCL constraints of the class History

```
context History inv:
  self.branch->size()>0
  self.branch
      ->forAll(b1, b2 | b1<>b2 implies b1.name<>b2.name)

context History::createRevision(base:Version,
                                 cp:ChangePackage):Version
  pre: base.branch.history = self
  pre: base.successors->select(link HistoryLink |
      link.getType()=RevisionLink)->size()=0
  post: base.successors->select(link HistoryLink |
      link.getType()=RevisionLink)->size()=1
  post: result = base.successors->select(link HistoryLink |
      link.getType()=RevisionLink)
```

```
context History::createBranch(v:Version, name:String)
  pre:   self.branch->forAll(b: Branch | b.name <> name)
  pre:   v.branch.history = self
  post: self.branch->size() = @pre.branch->size()+1

context History::createTag(v:Version, name:String):Tag
  pre:   self.branch.version.tag
      ->forAll(t: Tag | t.name <> name)
```

Branch

The Branch class represents a branch of concurrent development in the version space and is composed of all versions that are available in the branch. A Branch has a unique name attribute among all Branches of its History. A Branch has at least one Version instance, which has no associated Revision Link as predecessor. This instance is the initial Version of the Branch. If this instance has no predecessor at all, it is the initial Version of the History and thus, the Branch represents the main trunk of development . Otherwise, it has a Variant Link to a Version of a different Branch, and the Branch instance represents a variant of the trunk.

Listing 2.7: OCL constraints of the class Branch

```
context Branch inv:
  self.version->size()>0
  self.version->size() = self.version.predecessor
      ->select(l HistoryLink | l.getType()=RevisionLink)
      ->size()+1
```

Version

Versions represent the nodes in the version graph. The state of the project at a specific version can either be represented explicitly by a Project Data or implicitly by its position in the version graph and the appropriate deltas, represented by the Change Packages. The predecessor and successor Versions of a Version are identified by the subclass instances of the History Link class. These are the classes Revision Link and Variant Link. A Version can have at most one incoming and at most one outgoing Revision Link. If it has no incoming Revision Link, it is the initial version of a branch. If it has no outgoing Revision Link, it is the head revision of that branch. A Version can have an arbitrary number of outgoing Variant Links since it can have an arbitrary number of variants. However, a version can have at most one incoming Variant Link, in which case it is the initial revision of a new branch. A Version has a unique +revision:int attribute to identify the Version in a Branch and a +date attribute, holding the creation date of the Version.

Listing 2.8: OCL constraints of the class Version

```
context Version inv:
```

```
self.successor->forAll(h:HistoryLink |
    h.getType()=RevisionLink
    implies h.target.branch=self.branch)

self.successor->forAll(h:HistoryLink |
    h.getType()=VariantLink
    implies h.target.branch<>self.branch)
```

Tag

The Tag class identifies an user defined Version of the version graph by a user defined name. The name must be unique among all Tags of the History. The name attribute represents the name of the Tag.

Listing 2.9: OCL constraints of the class Tag

```
context Tag inv:
  Tag.allInstances()
    ->forAll(t1, t2 | t1<>t2 implies t1.name<>t2.name)
```

History Link

The History Link class is and abstract class, which is used to represent the edges in the version graph. The History Link represents the relationship between two different Version instances. It is a directed link class from the associated source to the associated target Version. The source and the target Versions must be different. The relation type is defined by the concrete subclasses Revision Link and Variant Link.

Listing 2.10: OCL constraints of the class HistoryLink

```
context HistoryLink inv:
  self.source <> self.target
```

Revision Link

The Revision Link class connects two different Versions from the same Branch. The target Version results from the source Version when applying the changes that are encapsulated in the associated Change Package. The associated Change Package represents the differences between the source and target Version.

Listing 2.11: OCL constraints of the class RevisionLink

```
context RevisionLink inv:
  self.source <> self.target
  self.source.branch = self.target.branch
```

Variant Link

The Variant Link class represents the relation between two different Versions from

two different Branches. Both Versions can be concurrently and independently changed, which results in the creation of new Versions and Revision Links in the related Branch. The Variant Link identifies the initial Version in a Branch that is based on a Version from a different Branch.

Listing 2.12: OCL constraints of the class VariantLink

```
context VariantLink inv:
  self.source <> self.target
  self.source.branch <> self.target.branch
```

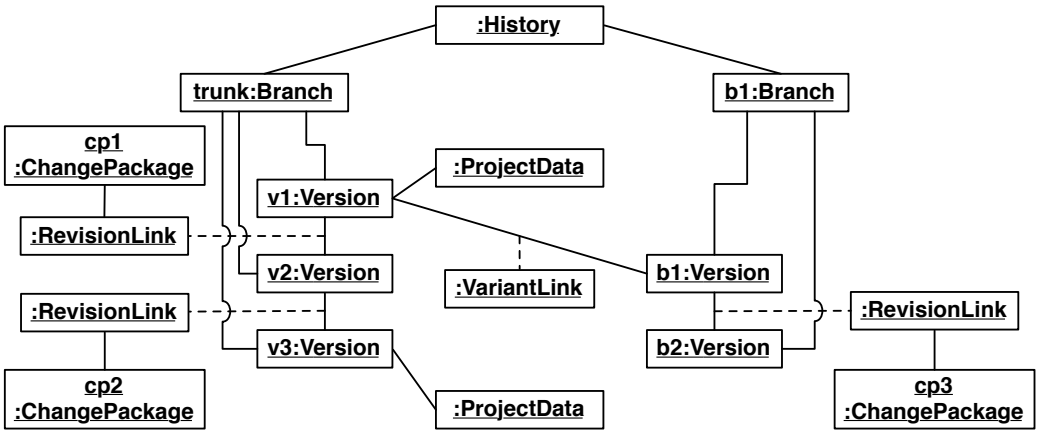


Figure 2.9: Example instance of the version object model

The UML object diagram shows an example instance of the version object model. The History consists of two branches, the trunk and the variant Branch b1. The trunk has three Versions. Version v1 is the initial version of the trunk and of the overall project, as it has not Revision Link or Variant Link as predecessor. Revision changes between the different versions are represented in the Change Packages associated to the Revision Links.

The following chapter describes the rationale-based unified software engineering model that extends the described meta-model. The model is separated into system models, collaboration models and the organizational models and describe their associations.

CHAPTER 3

The RUSE Model

This chapter presents the rationale-based unified software engineering model (RUSE model), which integrates system models, collaboration models and organizational models. The system models consists and integrates concepts from different modeling techniques. It covers feature modeling, use case modeling, object-oriented analysis, hazard modeling, requirements and system design. In addition, concepts for document models and diagrams of the RUSE model elements are provided. The collaboration models support informal communication, formal discussion models to capture rationale and a task model. The organizational model describes the project organization and integrates the participants with the system model and collaboration model.

All models are based on the extendable RUSE meta-model described in chapter 2. All classes of the RUSE model are subclasses of the Model Element class. The associations of the RUSE model are association classes that extend the Model Link class, except those associations that are marked with a «subelement» label. These associations are using the parent–subelement mechanism described in section 2.2.1. All attributes are realized with the field, list, and map mechanisms of the Model Element class. To reduce complexity from the class diagrams in the following, we do not show the classes of the RUSE meta-model again, except for those cases, in which we directly refer to them. Figure 3.1 shows an example. The class diagram on the top shows a model that is based on the RUSE meta-model. The simplified version of the model is shown at the bottom.

3.1 Organizational models

The RUSE model contains an organizational model consisting of Organizational Units that are either Participants or Teams. Teams in turn consist of other Organizational Units. Participants can be members of many Teams. An Organizational Unit has

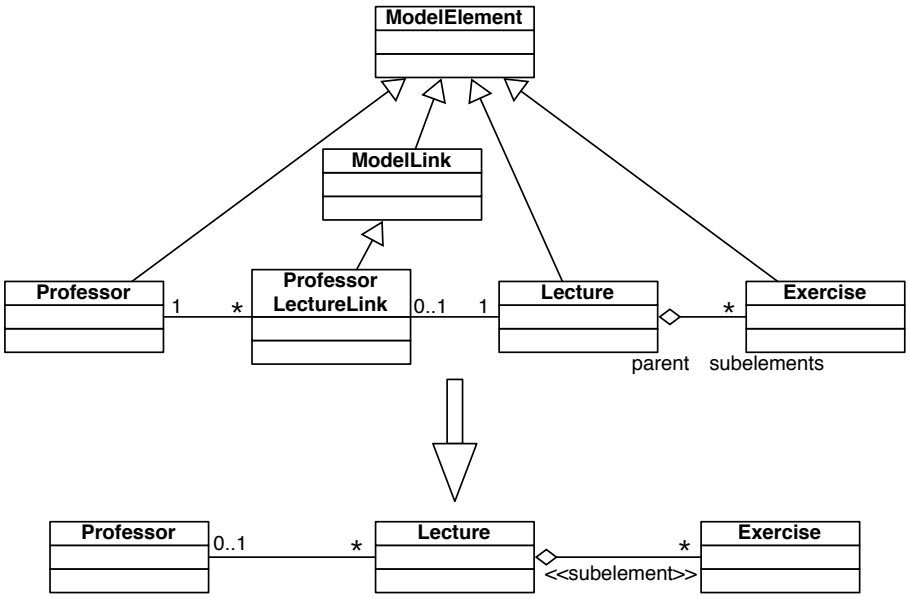


Figure 3.1: Example class diagram that is based on the RUSE meta-model.
The upper class diagram shows the model classes Professor, Lecture, and Exercise that extend the Model Element class of the RUSE meta-model. A Professor instance can be the lecturer of many Lectures, while a Lecture is associated to zero or one Professor. The association is realized by the ProfessorLectureLink class, which extends the Model Link class. In this model, a Lecture is composed of many Exercises. The lower class diagram shows the simplified notation of the model that we use in this chapter. The classes of the RUSE meta-model are hidden. The link class ProfessorLectureLink is replaced by a normal association. The association between the Lecture and the Exercise classes is marked with the «subelement» label to denote that the parent–subelement mechanism of the Model Element class is used.

attributes like name, address, phonenumber, and email, which provide the information to contact the Organizational Unit, and a description attribute.

We explicitly include the stakeholders of a project in the model. We add the Stakeholder class as a subclass of the Participant class. A stakeholder is a person or organization who has a certain interest and influence in the project. Typical project stakeholders are clients, project owners, or investors. The stakeholders are the main drivers of a project and their satisfaction can be used to define a successful project.

A Participant can be associated with at most one User, which is defined in section 2.2.1. The User defines the access rights of a person to the RUSE model in a Project. Typically, not every participant of a project is allowed to access the models. For instance, clients are normally not allowed to access the models during development. Only selected outcomes and work products are provided. Therefore, a Participant can be modeled in the organizational model but must not

be associated with a User. Conversely, each User who is allowed to access the Rationale-based unified software engineering model model is part of the project organization and must be included as Participant (see figure 3.2).

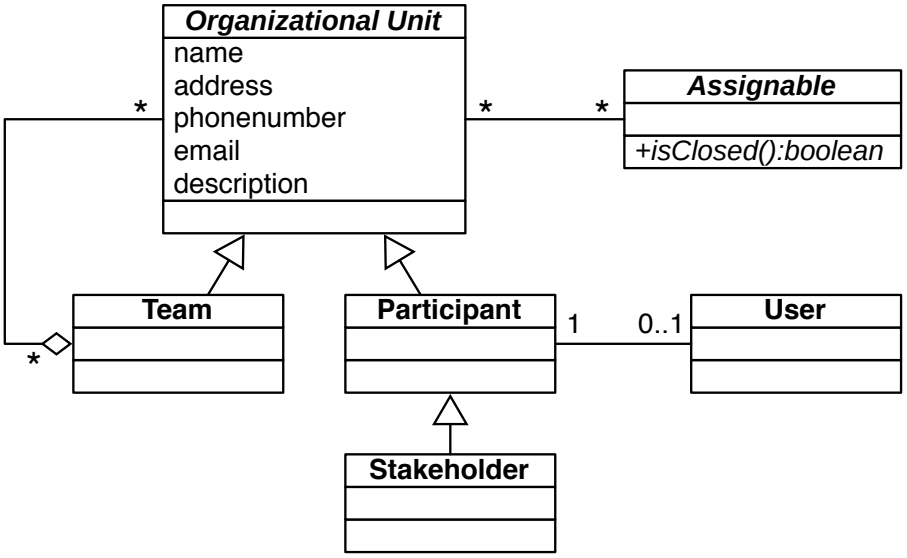


Figure 3.2: The organizational model

The UML class diagram shows the organizational model of the RUSE model. The organizational model represents the project in terms of Teams and Participants. A Participant associated with a User has access rights to the model.

Organizational Units are associated to many Assignables. The Assignable class is abstract and represents any items that need responsible Organizational Units. Subclasses include the Issue and the Work Item classes that are defined in section 3.3. An Assignable has a state that is either open or closed. The operation isClosed():boolean returns the state of the Assignable. Changing the state must be provided by the subclasses.

In difference to other approaches, the organizational model is represented in the same model as the system models. Thus, we can directly associate system model elements with the Participants and Teams and retrieve organizational information about authors and stakeholders of relevant elements.

3.2 System models

This section describes the system models of the RUSE model. It integrates different existing modeling techniques and is based on the unified requirements model presented in [13, 12]. We do not describe existing models in detail, but focus on the changes and extensions we made.

3.2.1 Stakeholder requirements

We explicitly start the description of the system models with project stakeholders. The stakeholders have different requests and expectations on a project, that we capture with stakeholder requests. The stakeholder requests are the initial elements for creating the project requirements. For requirements traceability it is essential to know the original stakeholders of a requirement [35, 69, 48, 47]. This knowledge is needed when negotiating about different and possibly contradicting requirements, as the original stakeholders have differing needs and influence within the project. Moreover, it is important to verify that each requirement is based on a stakeholder's request. Otherwise, it may be an orphan or "gold plated" requirement, whose implementation costs and resources have no financial justification.

The Stakeholder class is associated with many Stakeholder Requests. As many stakeholders such as clients do not access the system models of a project, the Stakeholder Requests are created and associated with the related Stakeholder by a responsible requirements engineer or analyst. A Stakeholder Request has a unique name, a description explaining required needs, and a status attribute that defines if the request is accepted or not.

If a stakeholder request is accepted, it will be associated with one or more features describing the stakeholder's needs in more detail, including possible system variability (see figure 3.4). According to the definition of Kang et al. (1990) [70] a feature is a property of a system that directly affects the end user:

> *"Feature: A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*

A Feature is detailed in any number of subfeatures that are mandatory, optional, or alternative. Mandatory subfeatures describe detailed aspects that the parent feature must support, while optional subfeatures may be selected when creating a concrete system from the feature model. Alternative subfeatures have a multiplicity similar to the UML multiplicity, which defines how many of the subfeatures must or may be selected [107]. A feature may be reused as subfeature by many other features; thus, a feature may have many parents. In addition to the hierarchical relationship, a constraint relation defines if a feature requires of conflicts with any other features.

A Feature directly results from a Stakeholder Request is called a *concept feature* and is represented within a feature diagram, which forms a tree. The concept feature is the root node of the feature diagram and all subfeatures are represented as child nodes. The hierarchical relationships mandatory, optional, and alternatives are represented by different edges between the nodes of the tree. A line with a filled cycle denotes the mandatory relationship, while the optional relationship is represented with a line, ending with an open cycle. An arc spanning two or more

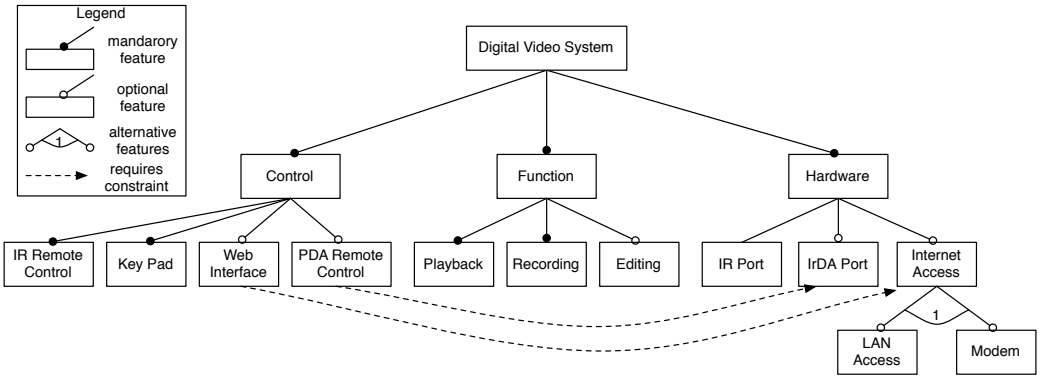


Figure 3.3: Feature diagram example

The figure shows an example feature diagram describing digital video disc recording product line. The diagram is adapted from [106] and is only shown partially.

edges of the feature nodes depicts a set of alternative features. The arc is annotated with the multiplicity of the alternative [124]. Figure 3.3 shows an example feature diagram that is adapted from Riebisch [106].

Feature models are able to describe the aspects, qualities or characteristics of a system and specially include variability modeling for system families. They focus on hierarchical decomposed system characteristics, but do not necessary include the end-users of the system. The structure of the feature model does not describe how a feature is used by the end-user, but rather provides a clear, unambiguous representation of the product and product line with all possible variations and combinations.

To describe the end-user interaction with a system, we use use case modeling. Use case modeling is a well-accepted technique to describe the interaction between end-users and the system as a textual flow of events. They represent functional requirements from an end-user perspective, as they describe what users can do with the system and how users interact with the system. Any parties outside the system that interact with the system are modeled as Actors. They may be human users or other external systems. By having a certain goal and requiring assistance of the system, an Actor initiates a Use Case, which in turn may involve many other participating Actors from which the system needs assistance to satisfy its goal [68, 93]. We separate the events flow into actor steps and system steps. Actor steps describe what an Actor does and the system steps describe the reaction or response of the system.

To combine the advantages from feature modeling and use case modeling in a single model, we introduce a new association that relates Features with Use Cases. The association specifies that the interaction represented by the Use Case, uses the associated Features. Therefore, the Use Case describes the Features from an

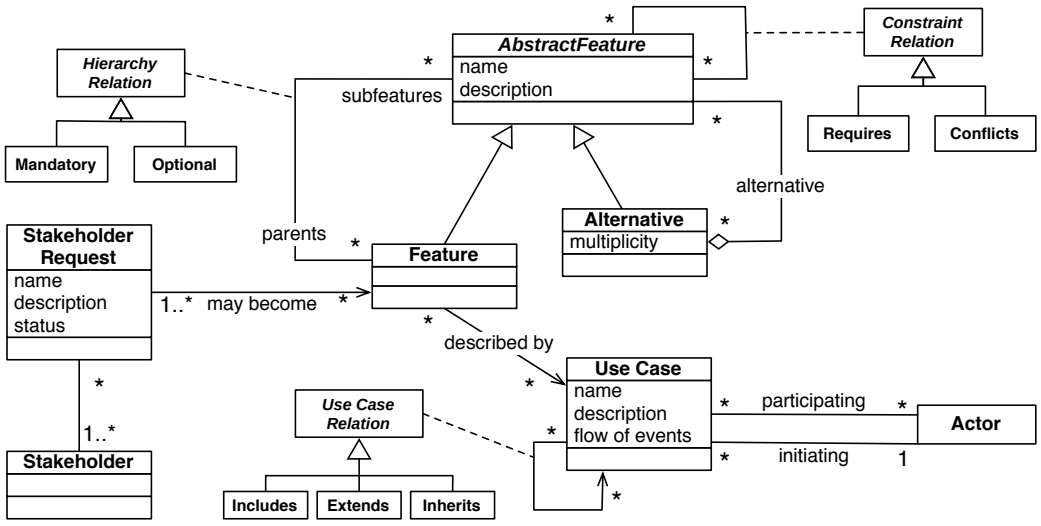


Figure 3.4: Stakeholder requirements

The UML class diagram shows the integration of Stakeholders and Stakeholder Requests, with Features and Use Cases. The classes describe the requirements of a project on a high level of abstraction, which is understandable by non-technical persons.

interactive end-user perspective. The Feature is either associated with the complete Use Case, or with a concrete event of the event flow. Thus, a Use Case can describe the interaction with the system including many different features.

A Feature should be associated with at least one Use Case. By traversing over the associations of a Feature and its Use Cases and between the associated Use Cases and their Actors, we identify the relevant Actors for the Feature. Features without Use Cases or Use Cases without Features should be subject of requirements reviews that identify whether the Features or Use Cases are really needed or if they should be revised. When looking at the features of some mobile phones, we question if really all features have a benefiting actor. Figure 3.4 shows the described stakeholder requirements model as a UML class diagram.

3.2.2 Requirements analysis

To explore and understand the problem domain, we apply an object-oriented requirements analysis as described in [19]. The analysis is based on the use case model and creates structural and a dynamic UML models and diagrams. To capture the relations and to maintain traceability between the stakeholder requirements and the analysis, we introduce new associations. By using Abbott's rules [1] for natural language analysis, participating objects are identified on use cases and are captured as class models. We create new participating objects associations be-

tween the Use Cases and the analysis classes. For clarifying and for describing the use cases in more detail and more formal, we use state, activity or sequence diagrams and create the expose in association between the Use Cases and the detailing diagrams.

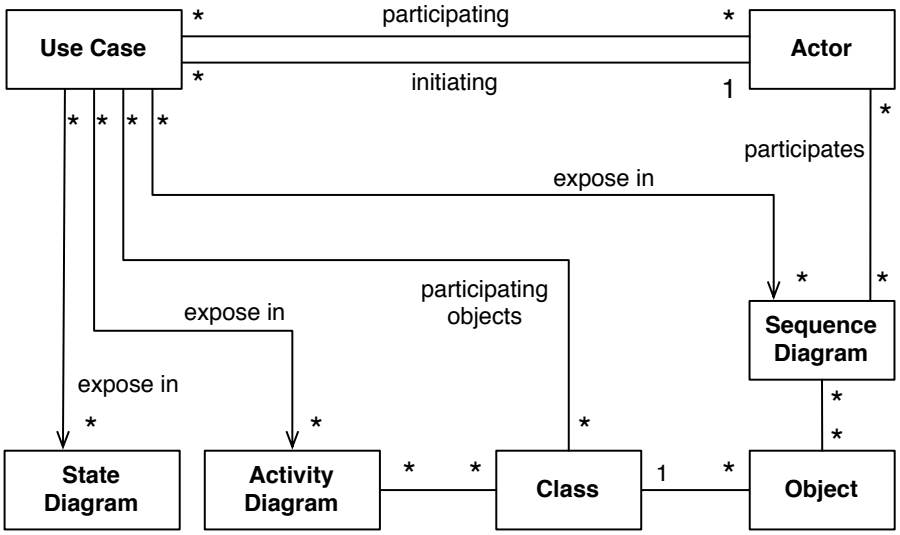


Figure 3.5: Requirements analysis

The UML class diagram shows the new associations between the Use Case and Actor classes and classes of the object-oriented requirements analysis.

The sequence diagrams instantiate the related use cases. All actors and all participating objects of a use case are also part of the instantiating sequence diagrams. The initiating actor of a use case also initiates the instantiating sequence diagrams and interact with boundary objects. Each event of the use case event flow is mapped to a message from the initiating actor to the boundary object or back. The requirements analyst can add new objects and messages to detail the internal control flow that describes how the goals of the use case are realized by the system. During this activity missing control and entity objects are identified. Whenever new messages between the actor and the boundary objects are created or deleted, the use case event flow gets automatically updated. Respectively, all changes of the use case event flow are automatically updated in the instantiating sequence diagrams [128].

Activity and state diagrams are used accordingly the sequence diagrams to expose a use case. Each event of the event flow is either represented as an activity or state. Activity and state diagrams are best suitable when focusing on alternative event flows. Analysis classes are used as input and output objects on activities.

We use the object-oriented analysis as described in [19] and do not describe all UML models in detail here. The focus is on capturing the relations between the analysis and the stakeholder requirements model as new associations. The as-

sociations represent the dependency traces and support consistency among these models when change occur. Potentially impacted elements are identified following the traces. Some changes can even result in additional cascaded automatic model changes, while other changes require a manually review of potentially impacted elements. Figure 3.5 shows the proposed associations between the Use Cases and the analysis model as a UML class diagram.

3.2.3 Detailed requirements

During detailed requirements analysis we inspect the higher level stakeholder requirements and create fine-grain functional and nonfunctional requirements. Each Requirement has a unique name and a description attribute. The granularity must be sufficiently fine that the requirement can be tested against any components realizing the requirement. Otherwise it must be refined until the required granularity is reached. Therefore, a requirement can be associated with many refining requirements.

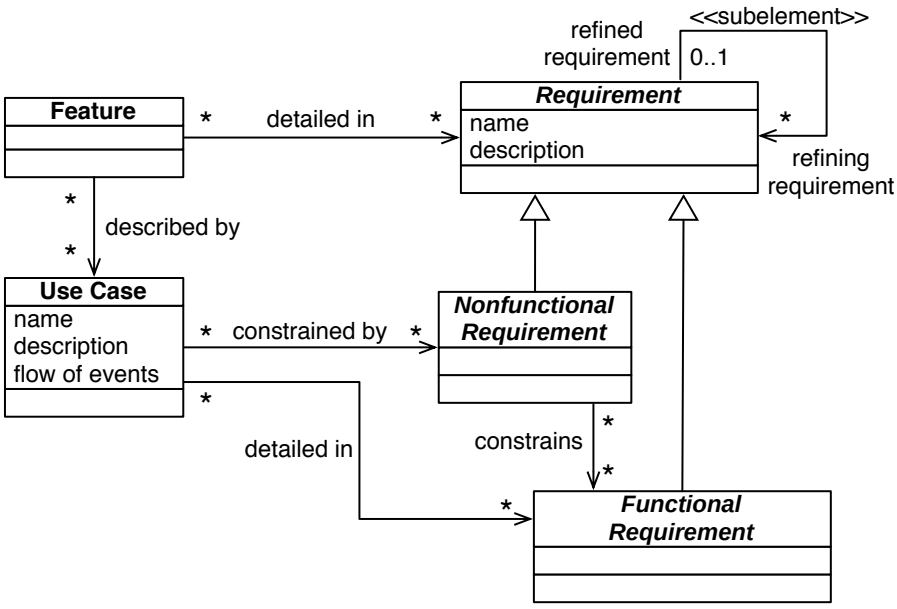


Figure 3.6: Detailed requirements

The UML class diagram shows the new associations between the stakeholder requirements consisting of Features and Use Cases and the detailed requirements consisting of hierarchical functional and nonfunctional requirements.

A Functional Requirement describes a required system function. A Functional Requirement can only be refined by other Functional Requirements. A nonfunctional requirement describes a property or quality of the system or of parts of the system and can only be refined by other Nonfunctional Requirements. A Nonfunctional Re-

quirement can constrain many Functional Requirements to indicate the quality of the associated functionality. The constrains association is in particular important for testing. For example, resting the Functional Requirement "The user must be able to login." is completely different if it is constrained by the Nonfunctional Requirement "The system must support 50000 users in parallel".

Listing 3.1: OCL constraints of the class Requirement

```
context Requirement inv:
  self.refining requirements
  ->forAll(r Requirement | r.getType()=self.getType())
```

A Functional Requirement must be based on at least one Feature or Use Case. A Nonfunctional Requirement is only based on Features, but can constrain many Use Cases. For instance, the Nonfunctional Requirement that a web form in a browser must completely appear within 4 seconds after being requested constrains any use cases where a user needs to see a form. The Nonfunctional Requirements might also be identified during use case modeling. Figure 3.6 shows the detailed requirements model and the associations to the Feature and Use Case classes.

3.2.4 Hazard analysis

During hazard analysis we identify potential Hazards for the end-users of the system under development. The Hazard class describes a potential harm or injury of a user, which might occur when using the system. It has a unique name, a description and a severity. As the users of the system are already represented as Actors in the use case model, we relate the Hazard class with a target association to many Actors. Conversely, an actor is the potentially target of many Hazards. When identifying a Hazard without finding any related target actor, then either the use case model is incomplete and needs to be refined, or the engineer has found a Hazard that is orthogonal to the system, which indicates that the Hazard is out of scope. In addition to the Hazard's target, any other analysis entities represented as classes (e.g. insurance company) may be involved with the Hazard. Thus, the Hazard can be related to many classes by using the involved entity association.

A Hazard may have many Causes, which describes the circumstances that leads to the Hazard. The Cause has a unique name, a description, a likelihood attribute, and an evaluation status. The evaluation status of a Cause depends on its likelihood and on the severity of the associated Hazard. It either indicates that mitigation of the Hazard or Cause is required. Otherwise, the current state of the related Hazard is acceptable. The target Actors of a Hazard are injured when a Cause of a Hazard gets triggered. As the Actors interaction is described by the use case model, we relate the Cause to the Use Case with a trigger association. In addition, a Cause may be associated with many hazardous elements. Hazardous elements are any model elements that describe the system under development and which are involved when a Cause gets triggered. We use the abstract super class System Model

Element to represent all possible model elements. For example, classes, components, controllers, or hardware nodes, which are realizing or which are involved with the triggering Use Case, are hazardous elements for the Cause.

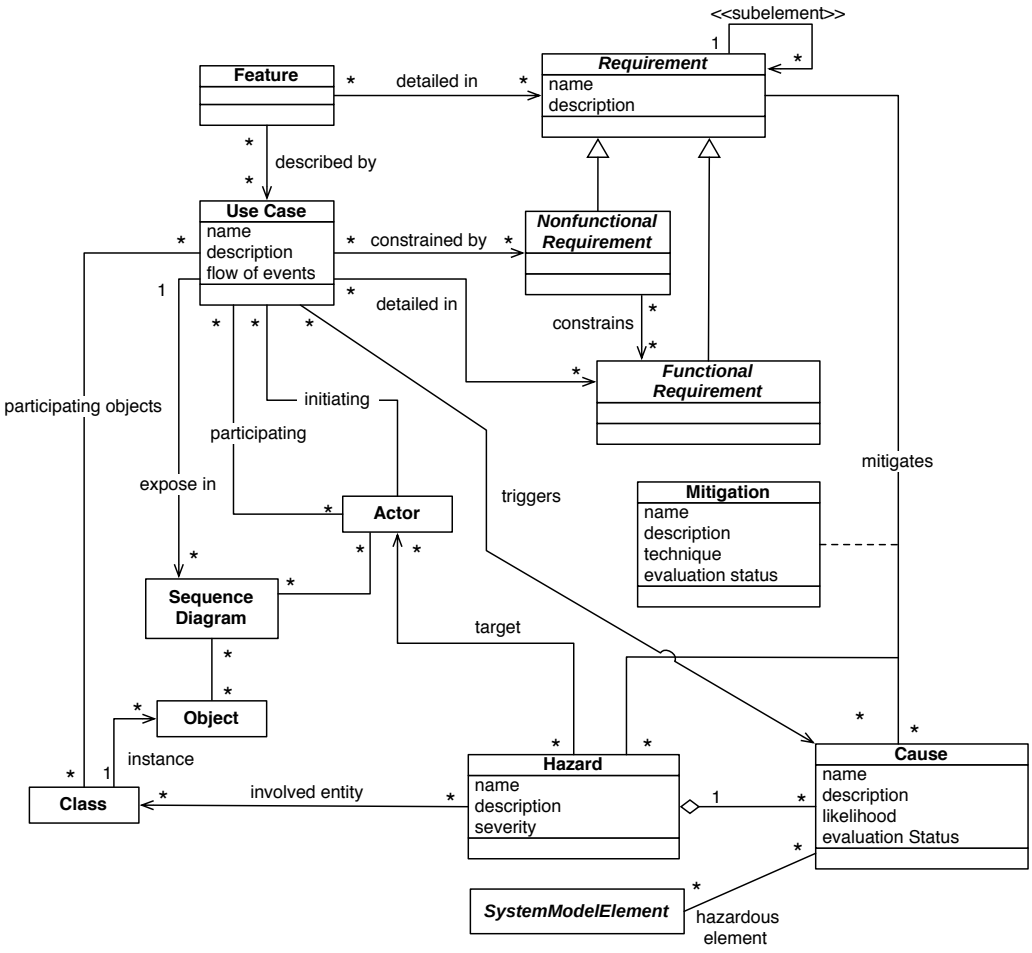


Figure 3.7: Hazard analysis

The UML class diagram shows the hazard analysis model including its associated classes from the stakeholder requirements, detailed requirements, and requirement analysis.

If Hazards including its Causes have an unacceptable risk or severity, mitigation of the Hazards or/and the Causes is needed until an acceptable risk and severity is reached. We mitigate the Hazards and Causes by creating new requirements that change the environment or the circumstances of the Hazards and Causes. A mitigation results in the reduction of the Hazard severity or of the likelihoods of the Causes. We use the association class Mitigation to capture the mitigation and to associate the mitigating requirements with the related Hazards and Causes. The class Mitigation has a unique name, a description, a mitigation technique, and an

evaluation status attribute. The evaluation status defines if the mitigation is accepted or not. Reviews with involved stakeholders is required before a mitigation is accepted and results in changing the severity of a Hazard, or changing the likelihood of a Cause. Figure 3.7 shows the hazard analysis model as a UML class diagram.

3.2.5 Diagrams

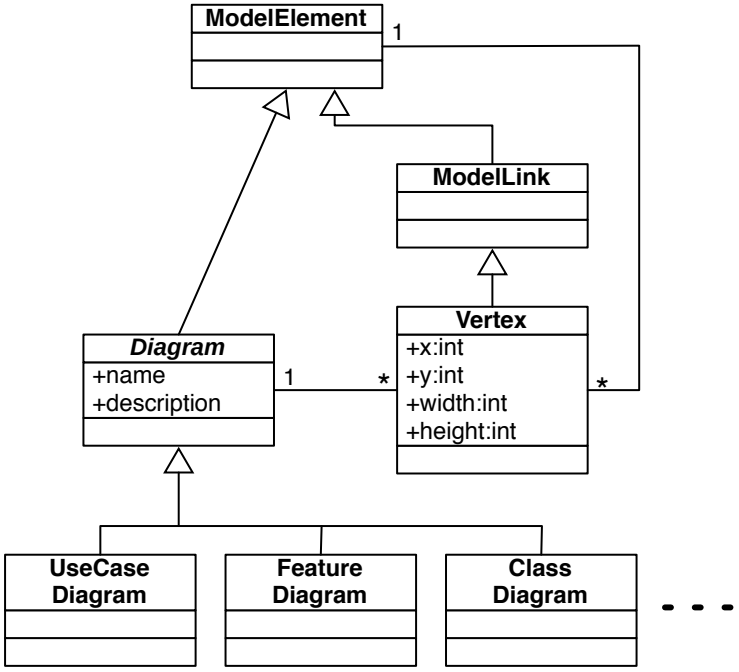


Figure 3.8: The diagram model

The UML class diagram in shows the generic diagram concept that enables the visualization of Model Elements in many diagrams. The Diagram class has a name and description attribute to identify and describe the Diagram instances. The Vertex class extends the Model Link class and associates the Model Elements with the Diagrams. Layout information is represented in the attributes of the Vertex. Subclasses such as the Use Case Diagram, Feature Diagram, or Class Diagram are used to ensure that only meaningful diagrams are created. Note that not all diagram subclasses are presented here.

Diagrams are mainly used to represent the described system models. For instance, Features are visualized in feature diagrams, Use Cases in use case diagrams, or classes in class diagrams. A system model element can be displayed in many diagrams, where each diagram has a different purpose. For example, classes are typically grouped into packages and a class diagram for each package is used to show all its classes. But a class diagram of a package is not able to include the associations between classes when one of the classes is not in the package. A

second diagram must be used to show the class with all its associations and associated classes. For each diagram, additional information such as layout information is needed to visualize the elements in diagrams. This information is related to the diagram and should not be part of the system model. Therefore, we explicitly separate the diagrams from the system models and provide a generic diagram model that enables to capture the required diagram information. The diagram model is shown if figure 3.8.

The diagram model supports consistency across all diagrams that contain the same Model Elements. For instance, when a Model Element is renamed, all diagrams reflect the change immediately, as they refer to the same element. When a Model Element is deleted, all associated Vertex links are deleted too, and the element is automatically removed from all diagrams. Moreover, starting from a given Model Element, all Diagrams can be identified by traversing over the associated Vertex instances.

3.2.6 Document model

In software development projects, documents are used to capture the outcome of an activity. For instance, the requirements analysis is documented in the Requirements Analysis Document (RAD) or the system design is documented in the System Design Document (SDD) [19]. In general, they consist of the activity related models and text. The models are created and maintained in modeling tools, while the documents are written with word processing tools. The models must be exported either into a graphical diagram or into a textual representation, before integrating them into the documents. Whenever a model is changed, it must be exported again, and the related documents must be updated. Thus, the documents get outdated very fast and manual effort is required to keep the documents and models consistent.

To overcome these problems, we integrate a document model into the RUSE model. All Model Elements are organized and viewed in the context of documents or as a result of filters. A document is defined in terms of sections and subsections, each containing text, diagrams, or a filter. The filter is used to attach matching Model Elements to the sections. A filter is defined as a class of element (e.g., Use Case) and an optional number of property name and values (e.g., “priority = high” or “planned for release = 2”). Documents and filters are themselves Model Elements and can be customized for each Project. Figure 3.9 shows the classes of the document model.

The filter mechanism enables that a single Model Element can be part of many documents or sections. When the Model Element is changed, all documents reflect the change immediately as they refer to the same element. Moreover, when the state of a Model Element is changed, it can dynamically appear in or disappear from a document, depending on the filter settings. This mechanism supports con-

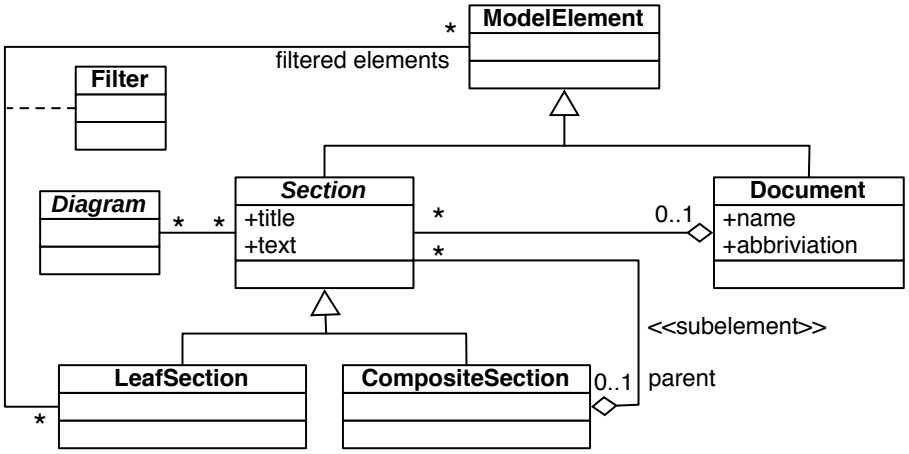


Figure 3.9: Document model

The UML class diagram shows the document model of the RUSE model. The Document class represent documents and has a name and a abbreviation attribute. The Document extends the Model Element and is composed of many Sections, which are either Composite Sections or Leaf Sections. A Section consists of a title, text and many Diagrams. The Composite Sections are used to realize a section hierarchy in which the Leaf Sections represent the leafs. Leaf Sections have a filter that is used to integrate any Model Elements in the section.

sistency across all documents and their included models. When the system models are changed, the documents do not require a manual change or update.

At the beginning of a project an initial set of documents, consisting of the empty sections and their filters, enables a template-based development. They can be changed during a project and may serve as new templates for subsequent projects. To enable collaboration with external stakeholders, the documents including all models and diagrams can be exported into files of different formats, such as PDF or RTF.

3.3 Collaboration Models

Capturing collaboration artifacts can help supporting future changes, such as finding the human source of a feature or a nonfunctional requirements and identifying related model elements, as indicated by studies about pretraceability [47] and rationale [71, 37]. The challenge is in making such collaboration capture practical and usable. To address this challenge we integrate the collaboration model into the RUSE model, so that collaborating about the system models can be supported in the same environment as developing the system model. Project participants are not required to switch their context and the collaboration artifacts can be captured. We provides an annotation mechanism that strikes a balance between

spontaneous, informal collaboration and formal, long-term rationale capture. We initially published the annotation mechanism in [129].

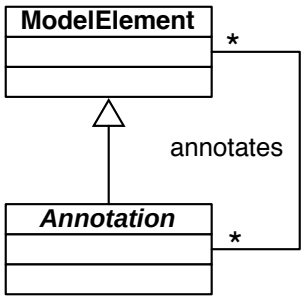


Figure 3.10: Annotation model
The UML class diagrams shows the Annotation class, which extends the Model Element class. Annotations can be associated to many Model Elements and are treated as first class objects.

In the RUSE model, users collaborate by linking collaboration artifacts, called Annotations, to Model Elements. Comment, Issue and Work Item are the main subclasses of the Annotation class and are described below. The Annotation class inherits in turn from the Model Element class (see figure 3.10). Consequently, annotations have the same importance as system model elements and, unlike in other tools, are treated as first class objects. A single Model Element can be annotated by many Annotations and a single Annotation can be linked to many Model Elements. Therefore, Annotations can be used to represent complex relationships, for example, connecting system elements that are not directly linked. As the collaboration artifacts are directly associated with system models, the collaboration context is always provided. Conversely, all collaboration artifacts can be identified for a given system model element. The traces across system models and collaboration artifacts are captured and maintained.

3.3.1 Informal communication

Comments are an informal and unstructured way for project participants to communicate, similar to posts in a newsgroup. Project participants can reply to existing comments, initiating discussion threads. Unlike in newsgroups, the RUSE model Comments and their replies can annotate and refer to any other Model Elements, including system model elements and other Annotations. The annotated Model Elements provide the detailed context for the discussion. Conversely, the discussions of a given Model Element can be identified. Figure 3.11 shows the UML class model supporting informal communication.

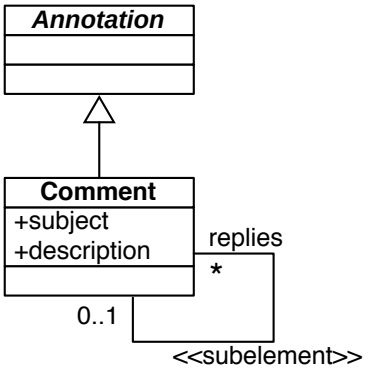


Figure 3.11: Informal communication

The UML class diagram shows the class Comment, which extends the Annotation class. A Comment can have many replies and represent a discussion thread. A Comment has a subject and a description attribute.

3.3.2 Issue model

While comments are suited for light-weight, short-term collaboration, such as requesting clarifications or indicating superficial problems, they are not sufficient for long-running or complex design discussions involving conflicting criteria and trade-offs. To support these types of discussions, the RUSE model provides an issue model that is based on QOC [80], including Proposals, Criteria, Assessments, and Resolutions (see figure 3.12). We chose QOC instead of the more popular IBIS model [25] because we observed that users often reverse engineer issue models from informal or threaded discussions as opposed to structuring them on the fly.

Issues represent needs to be solved for the development process to proceed. Issues can indicate a design issue, a request for clarification, or a problem resulting from a possible defect. An important part of the rationale is a description of the specific Issue that is being solved. Issues are usually phrased as questions. The Issue class extends the Annotation and Assignable classes. Thus, an Issue can be attached to any problem related system model elements. In addition, it can be assigned to Organizational Units that are responsible for resolving the Issue. Issues are closed when they are associated with a Resolution. They can be reopened again, which results in relating the previous Resolution instance with a contested association.

Proposals are possible solutions that could address the Issues under consideration. These include Proposals that were explored but discarded because they did not satisfy one or more associated Criterion instances.

The **Criterion** class represents desirable qualities that the selected Proposals should satisfy. In our model, the Criterion class is the super-class for nonfunctional requirements, system design goals, and test criteria. Thus, the qualities of the sys-

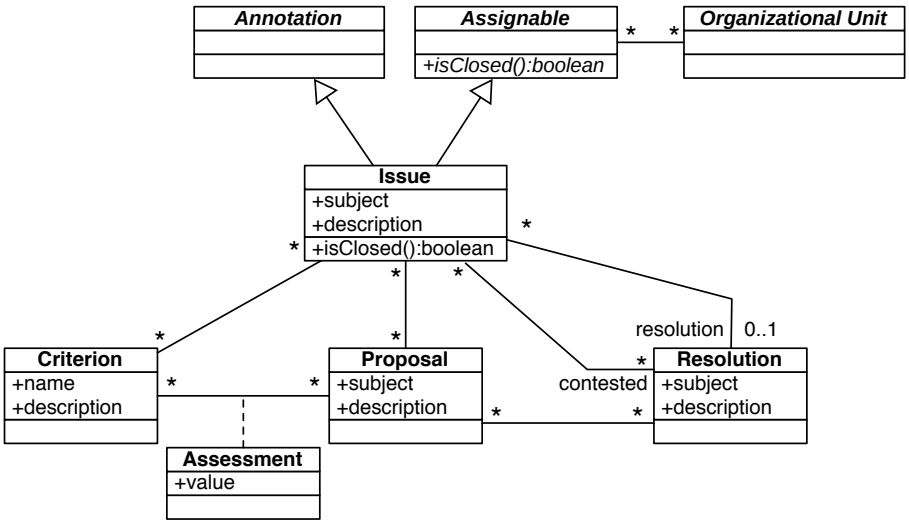


Figure 3.12: The issue model

The UML class diagram shows the issue model of the RUSE model. It consists of the Issue, Proposal, Criterion, Assessment, and Resolution classes. The model is based on QOC [80] and facilitates formal design discussions and the capturing of rationale.

tem under development can be used to evaluate the Proposals for system relevant Issues.

Assessments represent the evaluation of a single Proposal against a Criterion. An Assessment indicates whether a Proposal satisfies, helps, hurts, or violates a Criterion.

A **Resolution** represents the solution of Issues and is based on one or more Proposals. Contesting a Resolution results in the reopening of the associated Issues and relating the Issues and the Resolution with the contested association.

The issue model is used to capture the rationale behind decisions and thus, is capturing long term project knowledge. In existing approaches, the rationale is not captured explicitly. It is only available implicitly in the minds of the participants or in communication artifacts such as email, which is not accessible to all project participants. During stuff turnover, the rationale behind decisions gets lost. As the Issue class extends the Annotation class, the Issues can annotate any system model element. Therefore, the Issues can capture the rationale behind high level requirements such as Features, as well as the rationale behind analysis details, such as the attributes and operations of classes. Once an issue has been discussed and resolved, users can plan the resulting work as an aggregate of Work Items, that are described in the next section.

3.3.3 Task model

The RUSE model includes task model consisting of Work Items (see figure 3.13). A Work Item has a subject, a description, a status, a due date, and an estimate attribute. It extends the Annotation and the Assignable class. Thus, the Work Items can be annotated on any Model Elements that are related to the represented work and responsible Organizational Units can be assigned. The depends on association is used to relate depending Work Items. By using the «subelement» association, a Work Item can be decomposed into smaller Work Items, realizing a hierarchical checklist of work. The status attribute defines if a Work Item is open or closed, when it has no sub-Work Items. A Work Item containing sub-Work Items is closed, if all sub-Work Items are closed.

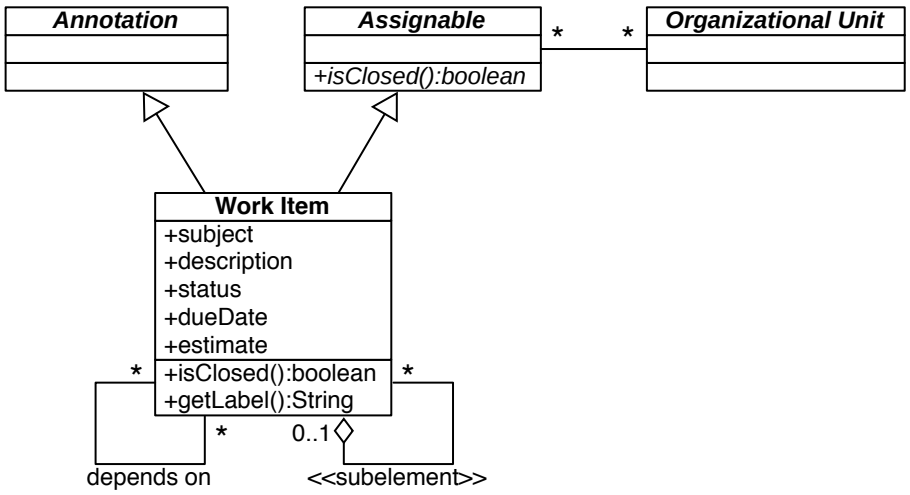


Figure 3.13: The task model

The UML class diagram shows the task model of the RUSE model. It consists of hierarchical Work Items that extends the Annotation and Assignable classes.

The operation getLabel():String returns an end-user readable label that depends on the state and structure of the Work Item. The operation returns “ToDo”, if the Work Item has no sub-Work Items, no defined due date, and no assigned Organizational Units. The operation returns “Action Item”, if the Work Item has no sub-Work Items, but a due date and assigned Organizational Units. It returns “Activity”, if the Work Item has sub-Work Items.

CHAPTER 4

RUSE Model usage and views

This chapter describes possible user interfaces and usages of the RUSE model. First we provide an overview of a graphical user interface window and how it is divided into separate areas that show the system, collaboration, and organizational models. Than we describe selected views in detail and how they meet the requirements provided in chapter 2. At the end we describe how the model and the user interfaces support traceability and awareness.

4.1 Views

As the RUSE model integrates the system models, the collaboration models, and the organizational models into one model, the project participants need only one user interface to access and manipulate all artifacts of a software development project. They can use the same interface to develop the required system models, collaborate and communicate with other participants over the models and explore the teams and participants of the project and identify their responsibilities and activities. Figure 4.1 shows the graphical user interface to the RUSE model, which is separated into four areas.

The upper left area is the navigation area that displays different configurable views into the RUSE model. A view is defined by a filter that filters for the Model Elements types, as well as for the state of their attributes. Defaults include views to show all Documents, Comments, Issues, Work Items, Diagrams, and an index view over all existing Model Elements. The users can define new views, for example a view containing all Work Items of a specific Team, or all Use Cases that are annotated by open Issues or Work Items.

The upper right area is a hyperlinked content area that is used to display any Model Element and the Diagrams in detail, including all their detailed attributes and relations to other Model Elements. The area enables the users to manipulate the Model Elements.

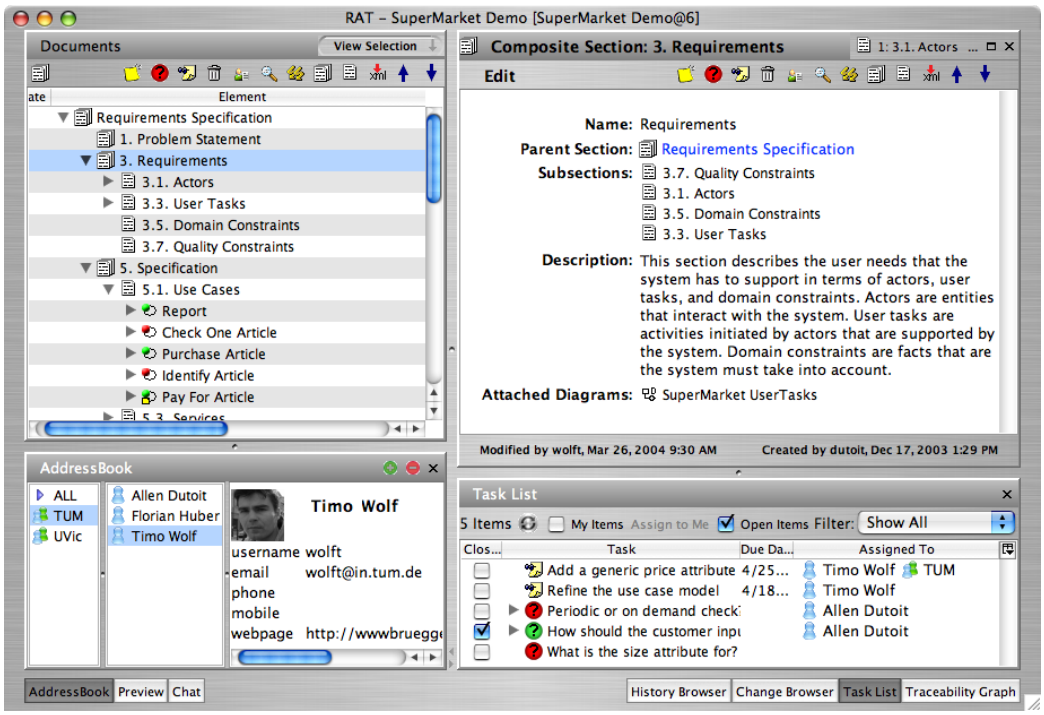


Figure 4.1: Overview of the graphical user interface to the RUSE model

The lower right area provides selectable user interfaces that supports frequent tasks of the users. They include the user interfaces for supporting traceability, for displaying the current user tasks, or for SCM functionality such as browsing through the project history and identifying recent changes.

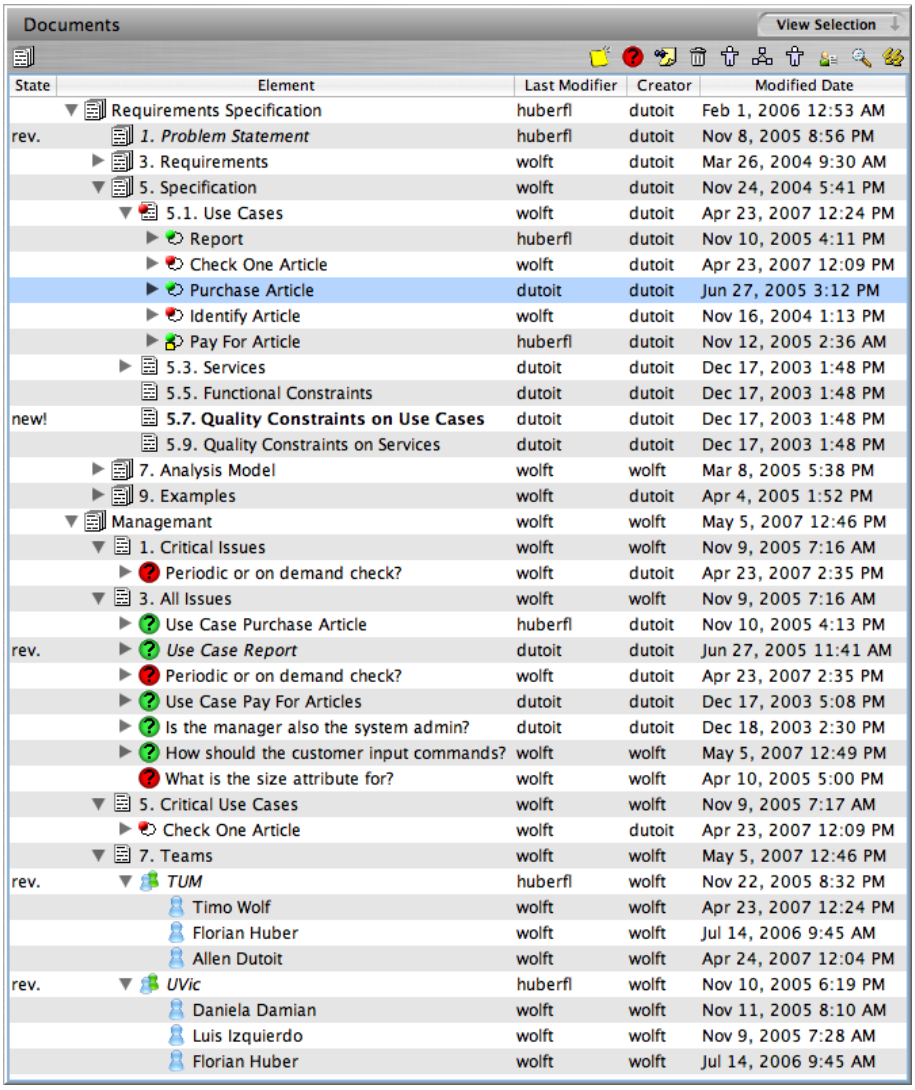
The lower left area provides selectable user interfaces including an address book that contains all Teams and Participants of the organizational model and a Chat that enables synchronous communication between the project participants.

4.1.1 Document view

The elements of the RUSE model are organized and viewed in the context of documents. A document has defined goals such as specifying the requirements, analyzing the problem domain, or documenting the software project management plan. As documents are part of the RUSE model and consisting of text, diagrams, and filtered Model Elements, they are used to group the elements of the RUSE model that are used to meet the goals of a document. Document templates are created at the beginning of a project and help to focus on the RUSE model elements of the document related activity.

In existing approaches documents are large binary files. Exploring the documents require the usage of the related word processing tool and to scroll through

large number of pages. Investigating multiple large documents simultaneously is almost impossible. If a model such as a diagram is reviewed in a document and needs to be changed, an external modeling tool needs to be used, the diagram needs to be changed, and the exported diagram image needs to be updated in all related documents.



State	Element	Last Modifier	Creator	Modified Date
	Requirements Specification	huberfl	dutoit	Feb 1, 2006 12:53 AM
rev.	*1. Problem Statement*	huberfl	dutoit	Nov 8, 2005 8:56 PM
	3. Requirements	wolft	dutoit	Mar 26, 2004 9:30 AM
	5. Specification	wolft	dutoit	Nov 24, 2004 5:41 PM
	5.1. Use Cases	wolft	dutoit	Apr 23, 2007 12:24 PM
	Report	huberfl	dutoit	Nov 10, 2005 4:11 PM
	Check One Article	wolft	dutoit	Apr 23, 2007 12:09 PM
	Purchase Article	dutoit	dutoit	Jun 27, 2005 3:12 PM
	Identify Article	wolft	dutoit	Nov 16, 2004 1:13 PM
	Pay For Article	huberfl	dutoit	Nov 12, 2005 2:36 AM
	5.3. Services	dutoit	dutoit	Dec 17, 2003 1:48 PM
	5.5. Functional Constraints	dutoit	dutoit	Dec 17, 2003 1:48 PM
new!	**5.7. Quality Constraints on Use Cases**	dutoit	dutoit	Dec 17, 2003 1:48 PM
	5.9. Quality Constraints on Services	dutoit	dutoit	Dec 17, 2003 1:48 PM
	7. Analysis Model	wolft	wolft	Mar 8, 2005 5:38 PM
	9. Examples	wolft	dutoit	Apr 4, 2005 1:52 PM
	Managemant	wolft	wolft	May 5, 2007 12:46 PM
	1. Critical Issues	wolft	wolft	Nov 9, 2005 7:16 AM
	Periodic or on demand check?	wolft	dutoit	Apr 23, 2007 2:35 PM
	3. All Issues	wolft	wolft	Nov 9, 2005 7:16 AM
	Use Case Purchase Article	huberfl	dutoit	Nov 10, 2005 4:13 PM
rev.	*Use Case Report*	dutoit	dutoit	Jun 27, 2005 11:41 AM
	Periodic or on demand check?	wolft	dutoit	Apr 23, 2007 2:35 PM
	Use Case Pay For Articles	dutoit	dutoit	Dec 17, 2003 5:08 PM
	Is the manager also the system admin?	dutoit	dutoit	Dec 18, 2003 2:30 PM
	How should the customer input commands?	wolft	wolft	May 5, 2007 12:49 PM
	What is the size attribute for?	wolft	wolft	Apr 10, 2005 5:00 PM
	5. Critical Use Cases	wolft	wolft	Nov 9, 2005 7:17 AM
	Check One Article	wolft	dutoit	Apr 23, 2007 12:09 PM
	7. Teams	wolft	wolft	May 5, 2007 12:46 PM
rev.	*TUM*	huberfl	wolft	Nov 22, 2005 8:32 PM
	Timo Wolf	wolft	wolft	Apr 23, 2007 12:24 PM
	Florian Huber	wolft	wolft	Jul 14, 2006 9:45 AM
	Allen Dutoit	wolft	wolft	Apr 24, 2007 12:04 PM
rev.	*UVic*	huberfl	wolft	Nov 10, 2005 6:19 PM
	Daniela Damian	wolft	wolft	Nov 11, 2005 8:10 AM
	Luis Izquierdo	wolft	wolft	Nov 9, 2005 7:28 AM
	Florian Huber	wolft	wolft	Jul 14, 2006 9:45 AM

Figure 4.2: Tree table document view

The documents of the RUSE model enable the exploration of multiple documents simultaneously. A tree table view of sections, their subsections, and the filtered Model Elements is used for a compact visualization. Figure 4.2 shows the tree table visualizing a “Requirements Specification” and a “Management” document. Sections of interest are expanded while others are collapsed. The creator, the last

modifier, and the last modification date is provided for each section, subsection, and filtered Model Element, as demanded by the *accountability* requirement from section 2.1. The realization of the *awareness* requirement enables that the elements, which have never been opened by the current user are flagged with a *new!* tag and are using a bold font. Revised elements, which have been modified after the current user opened them, are flagged with a *rev.* tag and use an italic font.

The filter mechanism of the documents enable Model Elements to be included in multiple documents and support dynamic document updating if the state of the elements change. For example, Figure 4.2 depicts the use case "Check One Article" in the context of the "Requirements Specification" document. The subsection "3.1 Use Cases" includes a filter displaying all use cases. The "Management" document in the same project contains a section that includes all high-priority use cases. When a new use case is created, it appears under the "3.1 Use Cases" subsection of the "Requirements Specification". When the priority of the use case "Check One Article" is changed from low to high, it dynamically appears in the "Management" document. When renaming an element such as a use case, both documents are kept consistent. Attached diagrams are also automatically updated, as they are just another view of the model. The filter mechanism of the Documents meets the *consistency of multiple model views* requirement from section 2.1 and uses the realization of the *support for fine-grained search and filter mechanisms* requirement. Each occurrence of a Model Element in a document is just a view. All views reference to the same Model Element and are consistent.

4.1.2 Visualizing collaboration artifacts

In section 3.3 we described that all collaboration artifacts are directly associated with the related system model elements, which are providing the context of the collaboration. To increase the awareness of ongoing collaboration such as discussions or open issues, we augment the icons of the related Model Elements with collaboration indicators. When Issues or Work Items are attached to a Model Element, the icon depicting the Model Element is augmented with a red ball for open and with a green ball for closed Issues or Work Items. Yellow notes are used when Comments are attached on Model Elements. The collaboration indicators quickly tell the user whether a model element is part of an ongoing discussion. The collaboration indicators visualize the realization of the *artifact integration* requirement from section 2.1.

Figure 4.3 is a section of the tree table view of the documents in figure 4.2 and shows the collaboration indicators on the icons of the use cases. The icon of the use case "Pay For Article" indicates that all related issues and work items are closed and comments are associated with the use case. The icon of the use case "Check One Article" indicates that the use case has either associated open issues or work items.

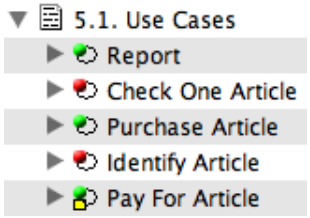
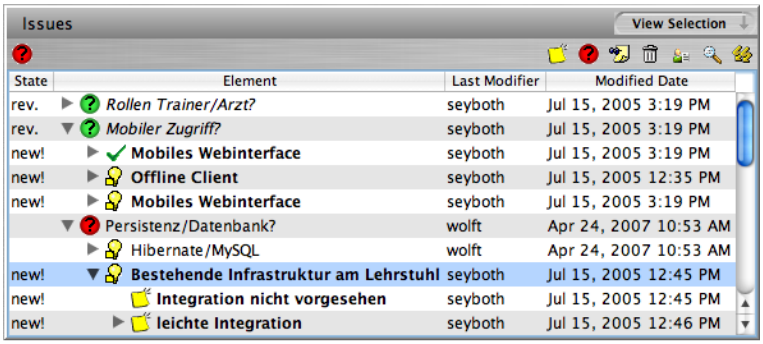


Figure 4.3: Collaboration indicators

The tree tables introduced above are also effective to give the users a compact view of the ongoing project collaboration. Figure 4.4 shows the issue models and discussion threads of a project in the tree table view. The views benefit from the realization of the *accountability* and *awareness* requirements. The user recognize the relevant participants of the collaboration Model Elements, as well as if the Model Elements are new for him, or have been revised after he read them.



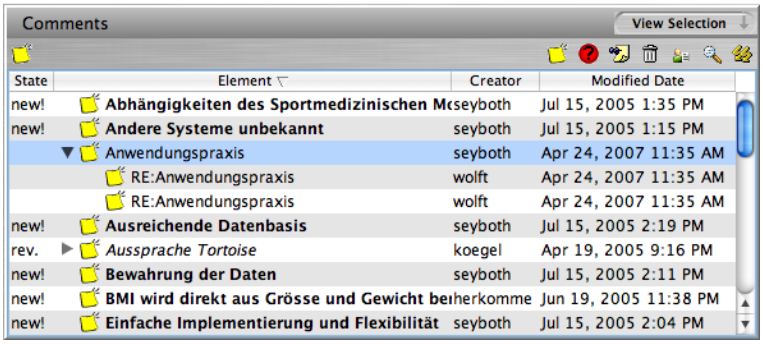


Figure 4.4: Tree table view of all issues and discussion threads.

4.1.3 Hyperlinked content area of Model Elements

When visualizing an element of the Rationale-based unified software engineering model model in detail, it's dependency traces can be traversed and all other related elements can be included as demanded by the *artifact integration* requirement. A set of modifications such as adding new Comments to related elements or creating new Issues are applicable without changing the view. In other approaches, the related elements are located in a different models and tools. A general linking concept between specified elements across different tools is not existent. URL hyperlinks to web-based applications are supported, but following the links require to change the view and the context of the user.

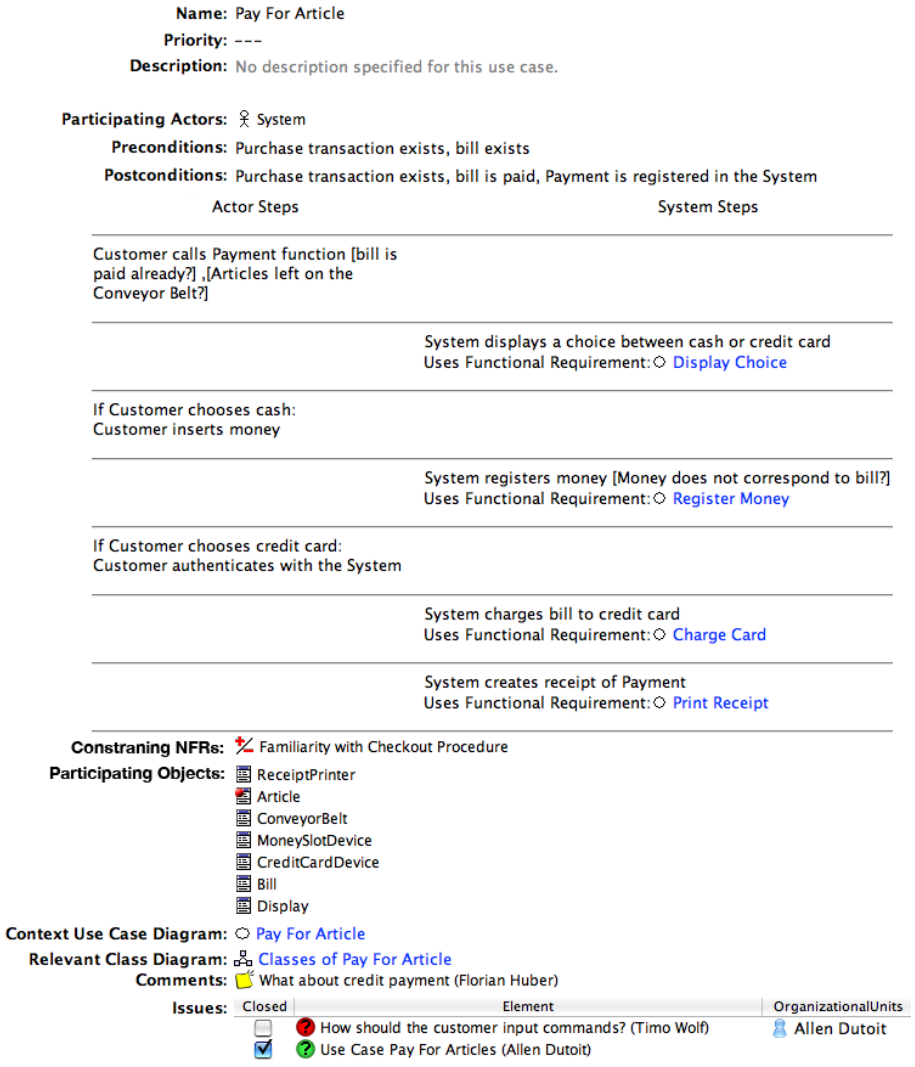


Figure 4.5: Example view of an use case

Figure 4.5 shows the details of the use case “Pay For Article”. It includes all use case relevant information such as the name, the participating actors, pre- and postconditions, or the flow of events. In addition, the flow of events include the associated Functional Requirements. A list of constraining Nonfunctional Requirements and all identified participating object are provided. The collaboration artifacts such as Comments or the open Issue “How should the customer input commands?” are depicted at the bottom of the view. The integration of the related Model Elements in the view of the Use Case follows the *consistency of multiple model views* requirement. Whenever the integrated elements are change, the Use Case view gets updated immediately. The All included elements are interactive. They can be directly modified by a provided context pop-up menu or can be opened and visualized in their own content view.

4.1.4 Diagram views

The diagram model of the RUSE model enables a diagram based visualization of all Model Elements. Standard diagrams for class models, use case models or feature diagrams are provided. All diagrams are always consistent with the model, as the diagrams are just a view of Model Elements. Changing a Model Element such as renaming a use case is reflected in all diagrams immediately, as demanded by the *consistency of multiple model views* requirement in section 2.1.

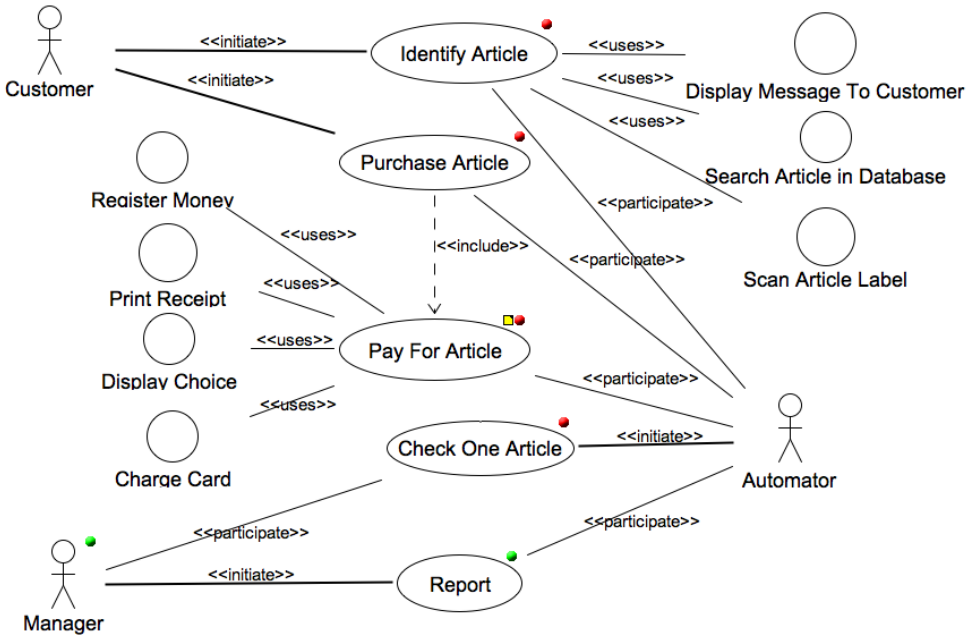


Figure 4.6: Use case diagram including functional requirements

The diagram concept enables extensibility for new types of diagrams. For instance, figure 4.6 shows a use case diagram including the Functional Requirements that are associated with the Use Cases. Use cases are represented with ovals and Functional Requirements with circles. The diagrams also visualize the collaboration indicators introduced in section 4.1.2. The diagrams provide a compact view of multiple Model Elements including their associated elements from different software engineering activities and the related collaboration models. The diagrams visualize the realization of the *artifact integration*.

4.1.5 Identifying work and participants

As demanded by the *artifact integration* requirement, all Work Items and Issues are included in the same model as the Organizational Units. Thus, project wide and user dependent task lists can be visualized. Figure 4.7 shows a task list including all Assignables and the assigned Participants. The items of the task list can be opened in the content area, which includes all item relevant annotated Model Elements, which provide the context of the item.

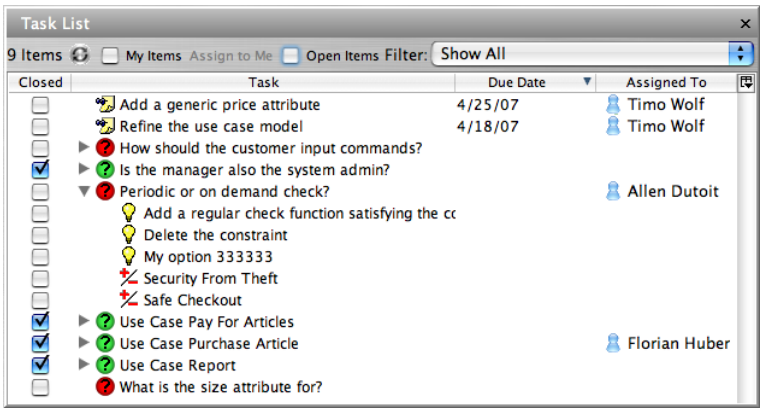


Figure 4.7: Task list view

According to the *accountability* requirement, the RUSE model captures all authors and modifiers of all Model Elements. Users can identify who created or modified the Model Elements. Following these information identifies the related Organizational Unit and opens it in an address book view. Figure 4.8 shows the address book with the Participant "Allen Dutoit". Conversely, a user can identify all Model Elements that were created or modified by a given Participant and find all associated Assignables. This mechanism increases the awareness of the current tasks of Participants and helps to identify on which Model Elements they work.

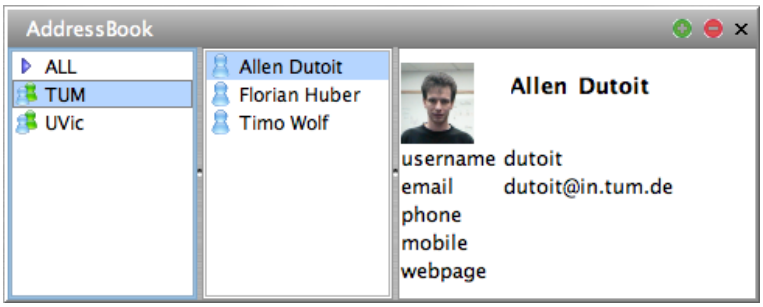


Figure 4.8: Address book view

4.2 Supporting traceability

The *traceability* requirement from section 2.1 demands for traceability links between all related system models, collaboration models and organizational models. We use the Model Link class and the inclusion of collaboration and organizational models in RUSE model to provide a uniform mechanism for supporting pre- and post-traceability. Let us consider three activities in which traceability is needed:

- **Identifying change impact.** Explicit links are used for tracing between the elements. These links are part of the model and are created by the users. For instance, links between Actors and Use Cases, Use Cases and Classes, or Use Cases and Functional Requirements are created during requirements, analysis, and detailed requirements. By following these links forward and backward, the user can assess the potential impact of a change.

- **Identifying stakeholders.** As Participants are also represented as Model Elements, the user can examine the author of a model element or trace from an Work Item to its assigned Organizational Units. For example, by examining who wrote a specific Nonfunctional Requirement and how it was used as a Criterion when resolving an Issue, the user can identify which qualities are critical for a Stakeholder.

- **Identifying implicit dependencies.** Annotations typically relate several Model Elements. As Annotations are also Model Elements, a user can trace over annotation links to find related elements. For example, a Feature can be referred in the discussion of a design Issue, involving a number of Nonfunctional Requirements. Even if the Feature is not explicitly linked to these Nonfunctional Requirements, the user will be able to find them by tracing through the Issue.

While traceability is conceptually simple to explain, it is not trivial to provide practical and usable traceability tools. On the one hand, users need a localized

view of elements to explore the immediately surrounding context. On the other hand, users need to view traces several levels deep to visualize a complete path between a pair of indirectly related elements. When addressing both of these requirements, the amount of information displayed quickly overloads the user We propose three graphical views to visualize and follow traceability links. All are focusing on the reduction of complexity, to support the users in daily work tasks.

4.2.1 Traceability tree

We propose a tree view to visualize transitive traceability dependencies. Each element is shown as a node in the tree and its direct related elements are shown as child nodes. The users can expand the nodes of interest and recursively expand the child nodes. Thus, the traceability paths over several elements are visualized. The users can reduce the number of elements by defining filters. For example, the user can define element types (e.g. use case, feature, or nonfunctional requirement) and link types to be included and excluded. Moreover, the user can define if incoming or outgoing traceability links are used to traverse to related elements.

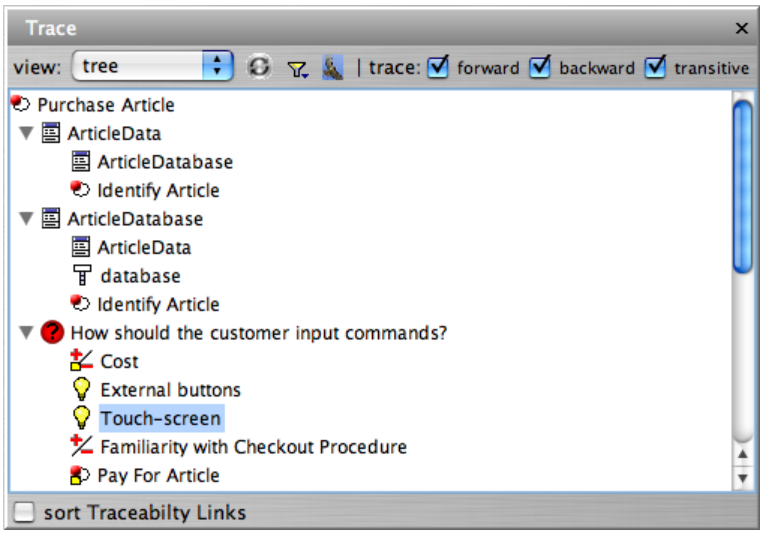
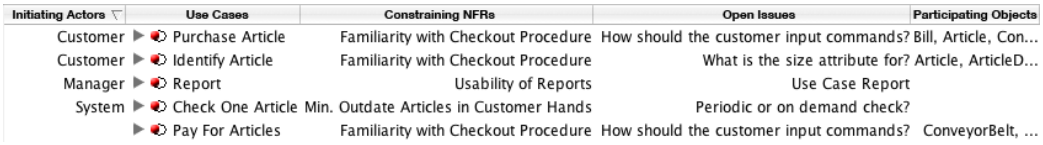


Figure 4.9: Example of the traceability tree view

Figure 4.9 shows an example traceability tree. The use case “Purchase Article” is expanded and the participating objects “Article Data” and “Article Database” are shown. The classes are also expanded and show the related use case “Identify Article”. The user can see a traceability dependency path from the use case “Purchase Article” over the participating classes to the use case “Identify Article”.

4.2.2 Traceability table

The second view focuses on all elements of the same type (e.g. use case) and supports the comparison of multiple elements. The elements are shown a table, with one element per row. The properties of the elements are displayed in the columns and also include the target elements of the traceability links. For instance, the figure 4.10 displays all use cases of an example project in the second column. Their related *Initiating Actors*, the *Constraining NFRs*, the *Open Issues*, and *Participating Object* from the requirements analysis are shown in the other columns. The user can open each traceability target element in its detailed view by double-clicking on its name. The included columns are configurable for each element type. The table rows are sortable by the columns and the table content can be searched and filtered. Hence, the tables are useful for inspection and review tasks. For example, a user can identify critical use cases by displaying all use cases and sorting for open issues. Alternatively, all use cases without any participating objects can be displayed.



Initiating Actors	Use Cases	Constraining NFRs	Open Issues	Participating Objects
Customer	Purchase Article	Familiarity with Checkout Procedure	How should the customer input commands?	Bill, Article, Con...
Customer	Identify Article	Familiarity with Checkout Procedure	What is the size attribute for?	Article, ArticleD...
Manager	Report	Usability of Reports	Use Case Report	
System	Check One Article	Min. Outdate Articles in Customer Hands	Periodic or on demand check?	
	Pay For Articles	Familiarity with Checkout Procedure	How should the customer input commands?	ConveyorBelt, ...

Figure 4.10: Example of the traceability table view

The figure shows the traceability table view, which contains use cases. The complexity of displaying traces is reduced by adding the trace targets into columns and using the label of the traceability links as column headers.

4.2.3 Traceability graph

The traceability graph focuses on one element and its surrounding context. The element under consideration is displayed at the center of the traceability graph while its directly related elements are displayed around it. Only links to and from the focused element are displayed, links among the other elements are hidden. For example, at the top of Figure 4.11, “Purchase Article” is the element under consideration. To differentiate quickly between types of relationships, the layout is separated into four areas: left, right, top and bottom. Contributor links are displayed on the left. Incoming system links are depicted on the left and outgoing system links on the right. Annotations (issues, comments, and tasks) attached to the system element are displayed below it while constraining elements (e.g., nonfunctional requirements) are displayed above. In other words, the horizontal axis displays system and organizational dependencies while the vertical axis displays rationale and collaboration artifacts. The user resets the focus of the graph

Figure 4.11: Interaction sequence of the traceability graph

The figure shows three steps of interacting with the traceability graph, in which the user clicks on a traced element. System model traces are displayed at the horizontal axis, while collaboration artifacts like comments and issues are displayed on the vertical axis. The creator and the last modifier of the focused element in the middle is always visible.

by clicking on any of the related elements, thus enabling a quick mechanism for exploring a full trace. To further reduce the amount of information displayed, the user can filter and search within the traceability graph. Double clicking on an element opens the element in a new detailed view that displays its properties and enables its modification. We initially published the concept in [130].

Figure 4.11 shows three steps of an user interaction sequence with the traceability graph. Each step shows the graph after the user clicks on a traced element, starting at the use case "Purchase Article". From the use case, he clicks on the issue "How should the customer input commands?" and ending at the nonfunctional requirement "Familiarity with Checkout Procedure". All elements related with the focused element, including the creator and the last modifier are shown in this example.

4.2.4 Capturing change impact

The proposed traceability views enable the exploration of dependent Model Elements, which might be impacted by a specific change. For example, if a Feature is the subject of change, depending Use Cases, their participating objects represented as Classes, and related Functional Requirements need to be reviewed and might be changed too. The requirements engineer responsible for changing the Feature might not have enough project knowledge or enough time to review and change all depending elements. Related project participants have to be informed about the change and need to review and change the model elements that are in their responsibility.

The traceability views support a mechanism to automatically create and annotate Work Items to all dependent elements. One initiating change Work Item is created and annotated on the element that is the subject of change and is assigned to the participant performing the change. One Work Item is created for each depending element and is added as subelement to the initiating change Work Item. Relevant project participants must be identified and assigned to the Work Items. The items appear in their task lists and notify the participants about the change and the review they have to perform. The overall change task is finished, when all elements are reviewed, changed, and all Work Items are marked as closed.

Figure 4.12 illustrates the mechanism with an UML object diagram. The Participant "Timo Wolf" changes the Feature "RFID Identification" and identified the depending Use Case "Identify Article" and its participating objects "Article" and "Article Label" by using the traceability tree. He generates the Work Item "Changed Feature 'RFID Identification' " and its sub-Work Items "Review to change of Feature 'RFID Identification' ". All items are automatically annotated to the related elements and assigned to responsible Participants.

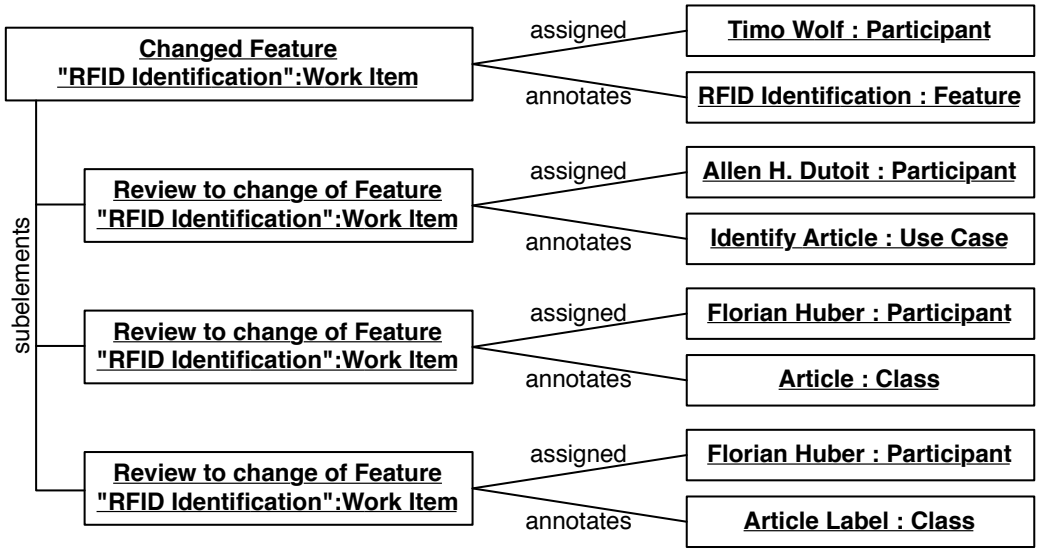


Figure 4.12: Object diagram of generated Work Items for capturing change

4.3 Supporting awareness

Within distributed projects, awareness of changes at other sites is key to reduce misunderstandings and redundant work and to increase trust across sites. Awareness helps to identify critical issues early and provides the opportunity to resolve issues in time [6, 36, 40]. In addition to capturing the activities of the project participants as required from the *awareness* requirement, mechanisms and services must use this information to automatically raise the awareness of relevant participants. Otherwise, the participants required inform themselves manually.

4.3.1 Subscribing to notifications

The RUSE model facilitates the subscription of users to receive notifications concerning changes about all Model Elements. Notifications range from changes to documents, system model elements (e.g. use cases, requirements, classes), to issues and work items. To avoid broadcasting all changes to all project participants, users can subscribe to the type of changes they want to be notified about. They can specify the model element classes of interest, such as use case, requirement or issue, resulting in an notification if any elements of these classes are created, modified, or deleted. They can also subscribe to changes in documents or in document sections, resulting in notifications when any model element within the document hierarchy changes. For example, if users choose a document that includes only high-priority use cases, they will be informed of all modifications to high-priority use cases, even if those use case did not yet exist at the time of subscription. Each

notification includes the type of change, the responsible person, the date and time, and a link to the modified element. Notifications are batched and sent to users at customizable intervals.

While notification subscription works well for system models (as developers usually have a well-defined area of responsibility), a different mechanism is needed for notifying changes in the collaboration model, as users are not necessarily aware of conversations that might be relevant to their work. To address this problem, notifications are sent to the last modifier of a model element, when annotations are attached to it. For example, if a user has modified a class, he will receive a notification if another user attaches an issue or a comment to this class. Similarly, a reply to an annotation will also trigger a notification to the author the initial annotation. In addition, users are notified if a Work Item is assigned to them or to their Team.

4.3.2 Disseminating changes

As not all stakeholders can be expected to access the RUSE model on a daily basis, the notification mechanism described above may not be sufficient to raise their awareness. To address this challenge, we propose to supports the dynamic creation of lists of URLs into the model. These lists can be included into web pages, to provide dynamic content about the state of the project. For example, a Wiki-based portal containing lists of all open Issues and open Work Items, organized by teams and sorted by date of modification and by responsible participants can be used by external stakeholders to get information of the project progress. Users can view the Wiki topics using a Web browser without being aware of the origin of the items.

CHAPTER 5

The Sysiphus environment

This chapter describes the system design of the Sysiphus [131] tool suite, which implements the RUSE Model and the RUSE Meta-Model described in the chapters 2 and 3. Beside the model and the meta-model, Sysiphus covers a server and several client components to access and manipulate the models of a project. In the year 2000, Allen H. Dutoit started the implementation of REQuest [38], a single web-application, which integrated use case modeling with rationale design. In 2002, we realized a first independent server component that implemented an initial version of the RUSE Meta-Model and added the client desktop application RAT [126]. We changed REQuest to use the RUSE Model and the new server component and published initial results in [128]. We constantly iterated over the system and applied it in academic and industrial contexts [21]. The applications are described and evaluated in chapter 6.

The structure of this chapter is based on the system design document template from [19, p. 283]. Section 5.1 describes the design goals, section 5.2 provides the subsystem decomposition and describes the subsystems. Section 5.3 maps the subsystems to hardware nodes. Persistent data management and access control are described in section 5.4 and 5.5. We conclude this chapter with the global control flow (section 5.6) and the boundary conditions (section 5.7) of Sysiphus.

5.1 Design goals

The design goals are qualities of the system. They build a consistent set of criteria that are used to evaluate design decisions and that are considered during the the subsystem decomposition, the realization of subsystems, and in the selection of off-the-shelf components [19, p. 248].

5.1.1 Performance Criteria

Response time The response time for all client applications of the system should be less than two seconds. Otherwise a progress indicator shall inform the user about the system activity. When possible, expensive computations shall be executed on the client nodes to reduce the load on the server node. Standard modifications of the RUSE Model, like changing the name and description of a use case, or adding a new association between classes must be performed within 1 second. The response time for modifying Model Elements must be constant *O(1)* and must not depend on the number of Model Elements in a Project. Boundary and initialization activities like loading or creating a project are not restricted in their response time. However, the response time should be at least linear to the number of Model Elements of a Project: *O(#Model Elements)*.

Memory usage We do not restrict the memory usage of the system to a concrete value. The used memory depends on the number of Projects and on the number of Model Elements in the Projects. The used memory must be at least linear to the number of all Model Elements of all loaded of Projects: *O(#Model Elements)*.

Network bandwidth The system should minimize the amount of used network bandwidth and compress the transfer data if possible. The network bandwidth for transferring a Meta Model Operation must be constant *O(1)*, and must not depend on the number of Model Elements of a Project. Transferring a complete Project must use linear bandwidth to the number of it's Model Elements.

Throughput The system should be able to handle up to 20 clients using an online workspace or up to 200 clients using an offline workspace simultaneously. Offline clients only require the repository server from time to time, while online clients continually send requests.

5.1.2 Dependability Criteria

Reliability The system should always preserve the state and the integrity of the RUSE model and the RUSE meta-model, including history data. Performing a change must result in an consistent and sound state of the data. In the case of a system error or crash, the data should be set to it's previous consistent state.

Robustness The system should validate all input provided by the user and display descriptive messages in the case of an error.

Fault tolerance The system should notify the user of any exceptions encountered during operation and cleanly shutdown the system in the face of unrecoverable errors. The data integrity should be preserved in the face of client and server node crashes. Client node crashes should impact the server node or any other client nodes.

5.1.3 Cost Criteria

Deployment cost Deployment cost should be low. Installing the system should be a task a normal user can perform and should not require the installation of any additional tools. End-user client applications should be executable without any configuration. The deployment of end-user client applications must conform to the platform typical installation process.

Upgrade cost Changing the RUSE model must be possible without changing or recompiling the meta-model. Adding new models or changing existing models must be possible without restarting the central repository containing the Projects.

Maintenance cost As the system is constantly being improved and new functionality is developed, the maintenance cost for bug fixes and extensions of the system must be low. Bug fixes in client applications shall not require a restart or even redeployment of the server or other clients. If, for any unknown reason, the persistent data integrity gets corrupted, experts should be able to fix the corrupted data manually.

Administration cost Administration cost should be low. A graphical user interface must be provided to support administrative tasks, like the creation of Projects and Users, and for setting the User access rights.

5.1.4 Maintenance Criteria

Extensibility The RUSE model implementation should be extensible without requiring any change on the meta-model or on the client applications. All subsystems must provide a well documented API, enabling reuse and the extension of the system with new subsystems, components, and applications.

Modifiability The system should be easily modifiable to be able to adapt the functionality of the system to new or changed requirements. Sysiphus will be freely distributable and changeable to fulfill its purpose as a research platform. It will be distributed under the GNU General Public License [44] in summer 2007. All off-the-shelf components, possibly used to realize subsystems, must not interfere with this licensing.

Adaptability The system must be highly adaptable to different application domains by extending the RUSE model. The extensions must not require any changes on other subsystems.

Portability Sysiphus must run on at least on Mac OS X, Windows and Linux. Java is used to implement the system, thus, all platforms that support the Java Runtime Environment can be used as target platform. Sysiphus should minimize the usage of operating system specific behavior. If operating system specific behavior is required, the interaction must be encapsulated by using the Bridge Pattern [46], and implementations for Mac OS X, Windows and Linux must be provided.

Readability All Java classes and its public methods must be documented with JavaDoc comments. Each package must provide a description, including it's goals and it's usage.

5.2 Subsystem decomposition

This section describes the hierarchical decomposition of Sysiphus into layers and it's subsystems. Each subsystem is described in detail, including its dependencies and main services. Sysiphus has an *open layer architecture* [110], consisting of the Element Store Layer, the Model Layer and the Client Application Layer (see figure 5.1). Each layer only uses services provided by its underlying layers and does not have dependencies to higher layers.

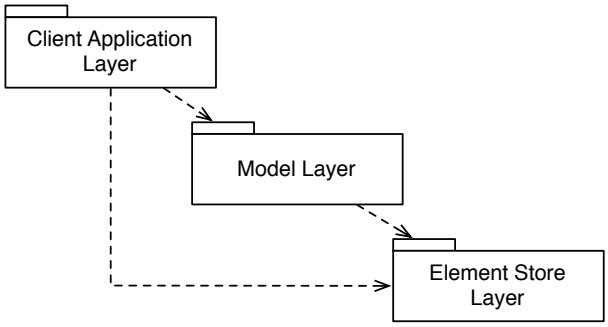


Figure 5.1: Open layer architecture of Sysiphus
The UML class diagram shows the layering of Sysiphus. The Client Application Layer contains the end-user applications to access and modify the RUSE Model, which is located in the Model Layer. The Element Store Layer provides the RUSE meta-model and a central sever repository. The repository contains the meta-model data that is accessible by distributed client applications.

Figure 5.2 shows the subsystem decomposition and subsystem dependencies of Sysiphus.

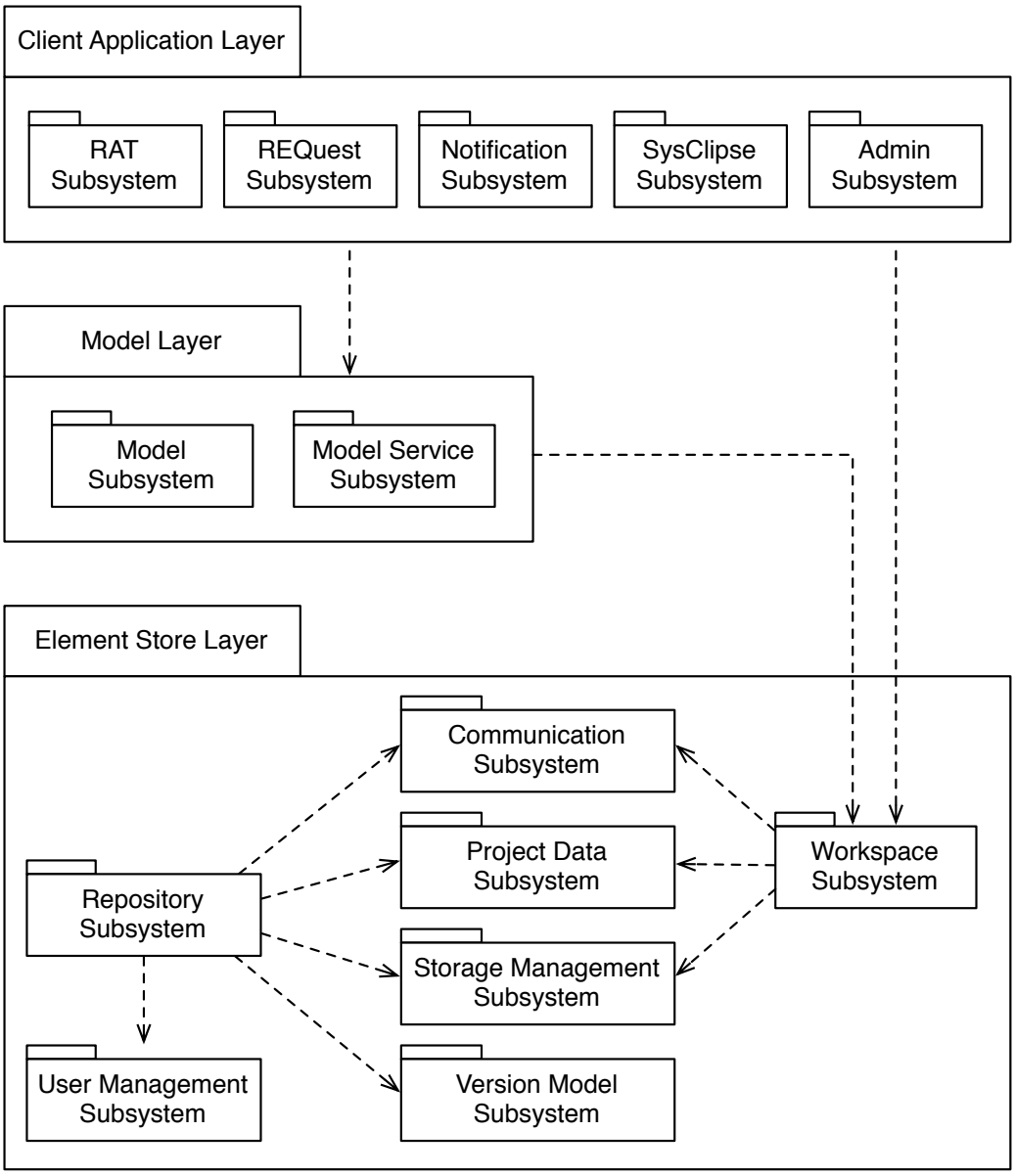


Figure 5.2: Sysiphus subsystem decomposition

The UML class diagram shows the layered architecture and the subsystem decomposition of Sysiphus. The Repository Subsystem and the Workspace Subsystem are the only subsystems of the Element Store Layer that are accessed directly by higher layers. The Repository Subsystem provides a Facade for the server repository containing the RUSE meta-model. To accomplish its task, it uses all other subsystems except for the Workspace Subsystem. The Workspace Subsystem provides access to the ProjectDataSubsystem and additional services to distributed client applications, located on the Client Application Layer. It uses the Communication Subsystem to access the Repository Subsystem. The Project Data Subsystem provides services to manipulate the RUSE meta-model. Persistency is handled by the Storage Management Subsystem. Access control is supported by the User Management Subsystem. The Model Layer provides the RUSE model and related services.

In the following, we describe the subsystems of these three layers, starting with the bottom layer.

5.2.1 The Element Store Layer

The subsystems in the Element Store Layer provide the Rationale-based unified software engineering model Meta-Model and underlying services for the Rationale-based unified software engineering model Model, including access control, persistent storage, and a central server repository that provides Project access to distributed client applications.

The Element Store Layer contains the following subsystems, which are discussed in the following subsections (see figure 5.2):

- Project Data Subsystem
- Version Model Subsystem
- Storage Management Subsystem
- User Management Subsystem
- Repository Subsystem
- Workspace Subsystem
- Communication Subsystem

5.2.1.1 The Project Data Subsystem

The Project Data Subsystem realizes the project data model described in section 2.2.1 and provides the services necessary to access and manipulate the project data model. In order to synchronize the project data models between multiple nodes without sending the whole data, the Project Data Subsystem provides the Change Package, the Model Operation and Meta Model Operations described in chapter 2.

The Project Data Subsystem is also capable of applying Change Packages to the data model. Applying Change Packages to the data model means to change the data model according to the encapsulated Meta Model Operations and handle possible exceptions. The Meta Model Operations manipulate the Model Elements, Model Links, its fields, lists, and maps, and the subelement structure, as discussed in 2.2.1.

5.2.1.2 The Version Model Subsystem

The Version Model Subsystem provides the version model as described in chapter 2. It provides services for creating, managing and retrieving versions and history information of a Project. The version model is independent of the project data model to satisfy the design goal of modifiability.

5.2.1.3 The Storage Management Subsystem

The Storage Management Subsystem provides persistency for project data model and the version object model. It provides services for storing, loading, and changing Projects and it's Model Elements on the persistent storage. To avoid storing the whole model for every change, the Storage Management Subsystem is notified when changes occur and is able to update the persistent data accordingly. The Storage Management Subsystem is configurable to either use a database management system, or a file based approach for realizing persistency.

5.2.1.4 The User Management Subsystem

The User Management Subsystem provides services for authentication, authorization and management of users. It uses a role and group based security model, as published in [28]. Each project has one or many assigned groups, containing users that are allowed to access the project. A user is in turn assigned to a number of groups. The group membership and the group roles define the access rights of the user for a project.

5.2.1.5 The Repository Subsystem

The Repository Subsystem provides a Facade to the shared server repository, containing the Projects and its Model Elements. It offers all service operations to retrieve and change the meta-model on a server node. In addition, the repository includes administrative services for project and user management. The Repository Subsystem provides the main controller for the shared server repository and delegates the invoked service operations to the related subsystems.

When the service operation for retrieving the project data model is requested by a workspace, the Repository Subsystem sends the data to the caller by using the Communication Subsystem. The invoking workspace can subscribe to receive synchronous push notifications about project changes. When the Repository Subsystem is requested to change the project data model according to a Change Package, it delegates the call to the Project Data Subsystem and the Version Model Subsystem to perform the necessary changes. The Storage Management Subsystem is notified to make the changes persistent. All subscribed workspaces are synchronously notified of the changes by sending a copy of the Change Package.

5.2.1.6 The Workspace Subsystem

The Workspace Subsystem provides the interface for accessing and manipulating the projects of the Repository Subsystem to higher layer subsystems, especially to the end-user client applications. It loads and maintains requested projects from the Repository Subsystem into the memory of a client node, thus enabling access to Model Elements without any network delay. The Workspace Subsystem delegates

all service calls, which manipulates the project data model, to the Project Data Subsystem, which realizes the requested changes. When Model Elements of the local workspace project get changed, it creates the necessary Meta Model Operations and Model Operations that describe the performed changes. The operations are collected in a Change Package. Whenever the Workspace Subsystem is requested by an upper layer to send the changes to the repository, it uses the Communication Subsystem to send the Change Package. The Workspace Subsystem supports two modes for manipulating Model Elements: a synchronous *online* and the asynchronous *offline* mode.

In online mode, the Workspace Subsystem ensures the synchronization of a project state and it's Model Elements on the client node with the state of the project from repository on a server node. Therefore, the workspace registers itself for pushed updates about project changes on the Repository Subsystem. The Workspace Subsystem sends any changes made on the client node to the repository immediately and receives any changes that were made by other clients. The online mode supports synchronous collaboration with multiple clients in a distributed environment.

In offline mode, the Workspace Subsystem isolates the client node from any changes of other clients. Instead of immediately synchronizing changes with the Repository Subsystem, it aggregates them in a Change Package. Synchronization of the local workspace project with the repository must be done by explicitly by calling a service operation.

The Workspace Subsystem is also responsible for conflict detection. Conflict detection is necessary when changes from the repository are integrated into a local workspace project, during an update or merge operation. The conflict detection service provides operations to detect conflicts between lists of Change Packages and to determine which changes are prerequisites for other changes.

5.2.1.7 The Communication Subsystem

The Communication Subsystem is responsible for transporting remote service calls, their corresponding data and results between the Workspace Subsystem and the Repository Subsystem. These subsystems may be located on different hardware nodes. To increase network performance, it uses compression when transporting the project data model. It uses socket connections to transport time critical data such as Change Packages to minimize network delay.

5.2.2 The Model Layer

The Model Layer provides the domain knowledge of software engineering artifacts and models. It consists of the Model Subsystem and the Model Service Subsystem. Both subsystems depend on the Element Store Layer, while the subsystems of the Element Store Layer are independent from the Model Layer. Changes and exten-

sions of the Model Layer subsystems can be done without impacting the subsystems of the Element Store Layer.

5.2.2.1 The Model Subsystem

The Model Subsystem provides the RUSE model as described in chapter 3. The subsystem relies on the services of the Project Data Subsystem from the Element Store Layer. The Project Data Subsystem is needed to realize the Rationale-based unified software engineering model Model conforming to the Rationale-based unified software engineering model meta-model. The Model Subsystem can be easily extended with new models. Every model instance of the Model Subsystem is represented by exactly one Model Element instance of the Project Data Subsystem. Note, that the Project Data Subsystem has no dependencies to the Model Subsystem, supporting the extensibility design goal.

5.2.2.2 The Model Service Subsystem

The Model Service Subsystem provides additional services on the Rationale-based unified software engineering model model. These services are used by several client applications and include the export of models to external PDF or RTF files, the support for JPEG or PNG image generation of diagrams, and services for text processing like the conversion of plain text to HTML and back. Furthermore, the subsystem provides user interface components that are used to view and manipulate models. The components are reused by different client applications to provide a common look and feel in Sysiphus.

5.2.3 The Client Application Layer

The Client Application Layer contains the subsystems of the Sysiphus client applications. The subsystems include the RAT Subsystem, the REQuest Subsystem, the Notification Subsystem, the SubClipse Subsystem, and the Admin Subsystem.

5.2.3.1 The RAT Subsystem

The RAT Subsystem provides the end-user desktop application RAT, a graphical user interface for accessing and manipulating the RUSE Model. RAT is comparable to other modeling CASE tools like the IBM Rational Software Modeler [64], supporting drag and drop interaction and the manipulations of diagrams. The RAT Subsystem provides the *online* and *offline* capabilities of the Workspace Subsystem to the end-user. The online mode enables a synchronous collaboration in a distributed environment. For instance, distributed design reviews and modifications of class models and diagrams are supported, in which all changes are propagated to all sites in real time. The offline mode enables the users to disconnect from the Repository Subsystem and to work offline without any network connectivity.

For instance, requirements engineers can create and change requirements at client site, while being offline. Changes are merged back into the repository when getting online again.

The RAT Subsystem provides a plugin mechanism that enables developer to create and add new plugins on runtime. The plugins get access to the loaded projects and can extend the functionality of RAT.

5.2.3.2 The REQuest Subsystem

The REQuest Subsystem provides the web-based client application REQuest, which provides a HTML-based hypertext view of the RUSE model, located in the Repository Subsystem. REQuest provides non-technical users access to the models and diagrams from a document-based perspective. REQuest is a server application in relation to the end-users, who access REQuest over HTTP by using any WWW Internet browser like Safari, Firefox or the Windows Internet Explorer.

5.2.3.3 The Notification Subsystem

The Notification Subsystem sends notification emails about user-relevant changes of a Project end-users. For instance, users get notified when they get assigned to Work Items or Issues. They also get notifications when collaboration artifacts like Comments or Issues are attached to models, they created or modified. The emails include an URL, displaying the relevant models in REQuest. In addition, the Notification Subsystem supports services to register for rule-based notifications, in which the users can specify about which changes they want to be notified.

5.2.3.4 The SubClipse Subsystem

The SubClipse Subsystem provides a framework for developing Eclipse plugins, which access the Repository Subsystem. The subsystem supports graphical user interface components for common tasks, like the login, logout or the selection of Projects. In addition, the subsystem provides Eclipse plugins to display user relevant Work Items and Issues, as well as displaying a graph, containing other project participants, which are assigned to related tasks.

5.2.3.5 The Admin Subsystem

The Admin Subsystem provides a graphical user interface for administrative tasks of the Repository Subsystem. The subsystem enables end-users with administrative access rights to create and delete projects, users, and groups and to define the group and user memberships and roles.

5.3 Hardware/software mapping

Sysiphus is a distributed application and its runtime components are deployed on several hardware nodes. This section describes the runtime components, the node types, and their prerequisites. Furthermore, we describe which subsystems are used in which components.

Sysiphus has a client–server architecture. Therefore, there are two types of nodes, namely client nodes and the server node. To satisfy the portability design goal, all subsystems of the Sysiphus are written in the Java programming language. Java features a Java Virtual Machine that provides a common application runtime on all kinds of different platforms. These include Windows, Linux and Mac OS X. Therefore, the Sysiphus components can deployed on any hardware node, independent from the operating system, as long as a Java Virtual Machine implementation available.

The runtime components of Sysiphus are listed below and an example deployment of the components to hardware nodes is shown in figure 5.3.

Element Store is the central server component of Sysiphus. It contains all subsystems of the Element Store Layer and provides access to the projects to distributed clients.

Element Store Admin is the application to administer the Element Store, realized by the Admin Subsystem.

RAT is the graphical desktop application to access and manipulate the RUSE model provided by the Element Store. RAT supports an online and offline mode. RAT uses the integrated Derby [4] database management system for persistent storage in offline mode. RAT is realized by the RAT Subsystem.

REQuest REQuest is a web-application that runs in an application server. It connects to the Element Store to access the projects containing the RUSE models and provides an HTML-based view on the models. REQuest gets accessed by browsers running on any client nodes.

Notification Service is a service application that connects to the Element Store to access the projects. It is realized by the Notification Subsystem.

Subclipse is an Eclipse plugin and provides the RUSE model in the environment of Eclipse. It is realized by the SubClipse Subsystem.

Figure 5.4 shows an example deployment of RAT and the Element Store on a client and server node. The diagram includes the subsystems and dependencies of the components.

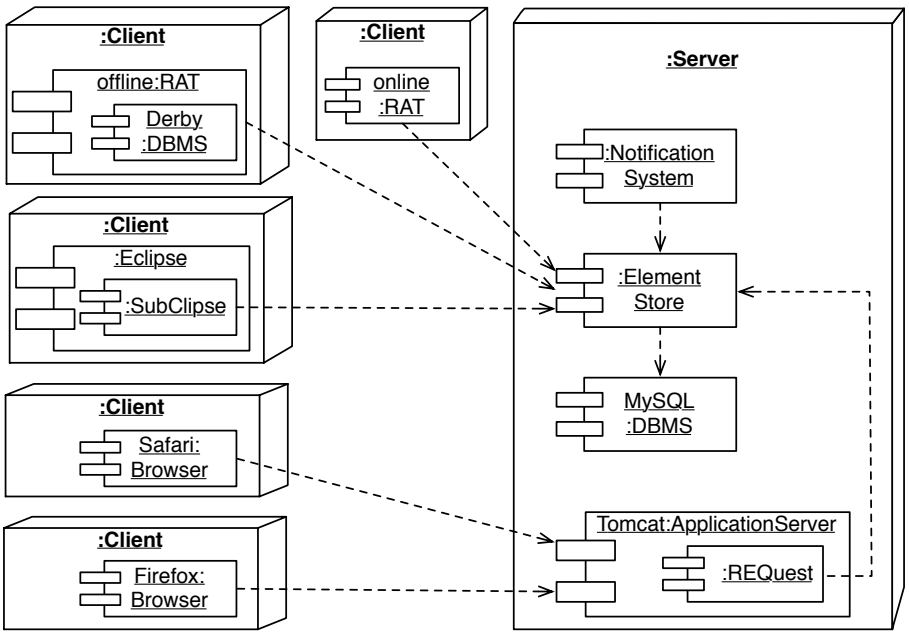


Figure 5.3: Deployment of the Sysiphus runtime components

The UML deployment diagram shows an example deployment of the runtime components of Sysiphus. The Element Store, the Notification Service and REQuest run on the same server node. The Element Store uses the database management system MySQL [87] for realizing persistent storage. REQuest runs in the Tomcat application server. Note, the components of Sysiphus running on the server node, could also be on separated nodes. The Element Store Admin is not shown in the diagram.

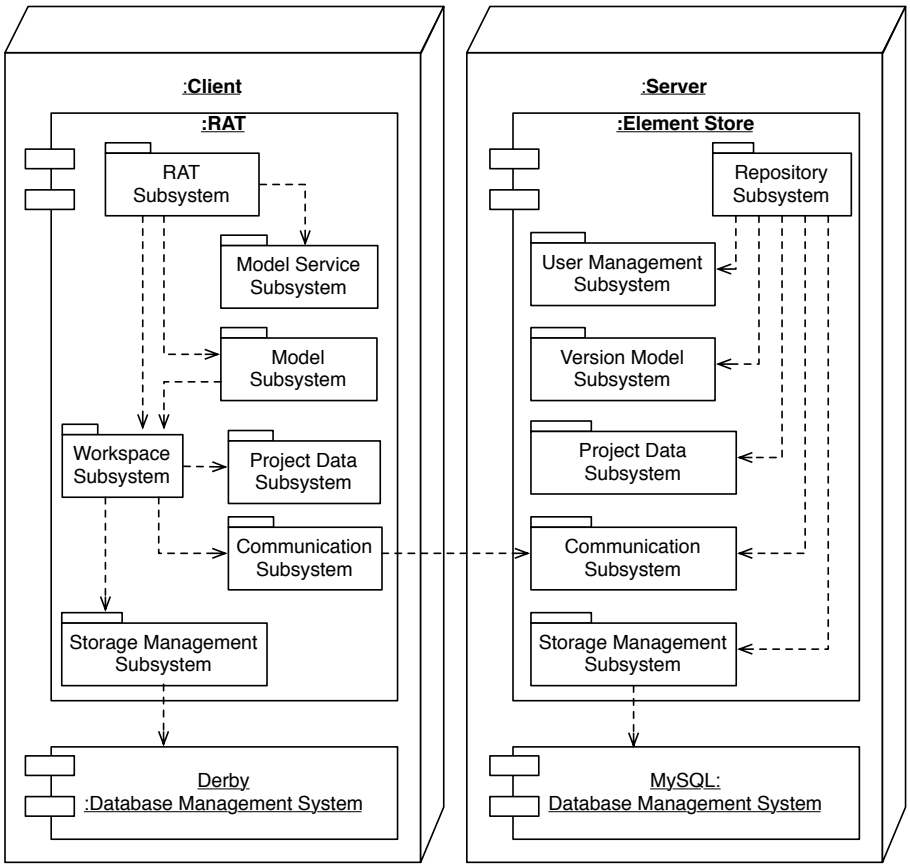


Figure 5.4: Deployment of RAT and the Element Store, including their subsystems

The UML deployment diagram shows RAT and the Element Store on a client and server node. The Element Store contains all subsystems of the Element Store Layer, except the Workspace Subsystem. It uses a database management system for persistent storage. RAT uses the RAT Subsystem of the the Client Application Layer, the Model Layer, and the Workspace Subsystem, the Project Data Subsystem, the Storage Management Subsystem, and the Communication Subsystem from the Element Store Layer. RAT uses the embedded database management system Derby [4], for persistent storage in the offline mode.

5.4 Persistent data management

Persistent data is present on both client and server nodes. The server node uses the Storage Management Subsystem to load, store and update persistent data. The Storage Management Subsystem is notified of changes to the project data model and updates the persistent data according to the Change Package, describing the change. The system supports a file storage mechanism, as well as storing the data into a database management system. The usage of a database management system is recommended in productive use, enabling the advantages of atomic transactions and regular backups. We use the MySQL [87] database management system for the Element Store on the server node.

Client applications with workspaces in offline mode need to store uncommitted changes and the project data model. Otherwise uncommitted changes and the current state of the data model would be lost. The Workspace Subsystem also uses the Storage Management Subsystem on client nodes. As the end-users should not need to install a database management system before using a client application like RAT, we use the Apache Derby database management system. Derby can be embedded in any Java application and needs no installation process.

5.5 Access control and security

This section describes how access to the data stored in the system is controlled and how these controls are enforced. In Sysiphus, only authenticated and authorized users can view and modify the RUSE model and its history. The Workspace Subsystems can not implement any effective access control, as they are running on different client nodes. The Repository Subsystem that is running on the server node is a facade for all server operations accessing the data of the Projects. Therefore, the Repository Subsystem enforces the access control and mistrusts by default any client. It provides service operations that enable the users to login and to retrieve a working session. The session must be provided in every service operation of the Repository Subsystem facade. It is used to test the user permission, before executing the operation. The user and session management is provided by the User Management Subsystem, used by the Repository Subsystem.

However, the Workspace Subsystem is caching the user and role information and is controlling access too. This enables the client applications to only provide valid operations to the users. Users can be immediately notified about any access violations on the local data, before the Element Store server would reject the committed changes due to missing privileges.

Sysiphus generally separates between an administrator role and a user role. The administrator role is required for all administrative operations, such as the creation an deletion of Projects and Users. In addition, the administrator is able to copy, archive and unarchive Projects and to set the access rights for roles and

Users. A user has access rights to Projects and is in general able to create, access, modify and delete Model Elements.

In addition, a mapping of Model Operations to Roles can be defined for each project. The mapping defines the access rights for a Model Operation of a RUSE model element and is stored on the Element Store server. When the server receives a change request in form of a Change Package, the User Management Subsystem validates all Model Operations of the Change Package against the provided access mapping. A new revision of the project data model is created, only if the user is allowed to execute all Model Operations. The mechanism enables a fine grained security and access rights definition on the operation base of a Rationale-based unified software engineering model model element.

5.6 Global control flow

This section describes the global control flow of the Sysiphus. Sysiphus uses an event-driven control flow [19, p. 276] to communicate between the Element Store server and different client applications. We describe the global control flow separately for the two workspace modes, the synchronous online and the asynchronous offline mode. Both control flows start with a client application (we use RAT) that modifies elements on the Model Layer. The Workspace Subsystem describes the changes to the meta-model in terms of creating Model Operations and Meta Model Operations.

5.6.1 Control Flow in Online Mode

The online mode facilitates synchronous collaboration between many distributed client applications, connected to the Element Store server. After changing the model and it's meta-model, the Workspace Subsystem uses the Communication Subsystem to send a Change Package, containing the generated Model Operations and Meta Model Operations to the server node. The Element Store server creates a new revision by changing the project data model and the version model according to the received Change Package. All changes made persistently by notifying the Storage Management Subsystem.

After successfully creating a new revision, the Repository Subsystem of the server node sends the Change Package to all other client applications, whose Workspace Subsystems registered for push updates on the modified project. By using the Project Data Subsystem, the affected client Workspace Subsystems modify the affected project data models, according to the received Change Package. The initiating sender client is blocked, until all client nodes are updated, and the Element Store acknowledges the change. Figure 5.5 shows a simplified sequence diagram of the control flow in online mode.

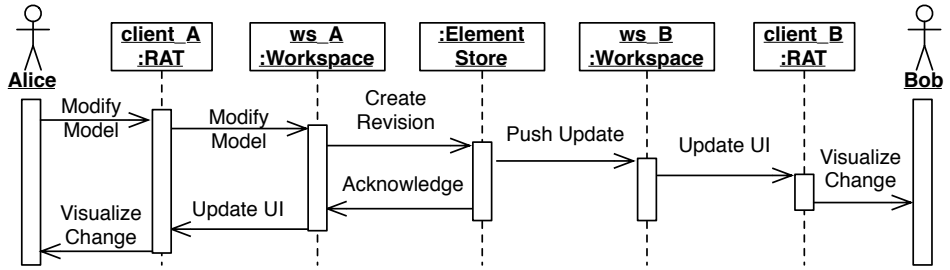


Figure 5.5: Control flow in online mode (UML sequence diagram)
The sequence diagram shows the two end-users Alice and Bob, both working with RAT on distributed hardware nodes. Both applications are connected to the Element Store, located on a central server node. After Alice modifies the model in her client application, the workspace sends the changes to the repository, which creates a new revision. The repository sends a push update, containing the changes to the online workspace used by Bob. The changes get applied and visualized at the RAT instance of Bob before Alice changes get acknowledged.

5.6.2 Control Flow in Offline Mode

In offline mode, the Workspace Subsystem stores the performed changes in a local Change Package, until an explicit commit operation is invoked on the workspace. When a commit operation is invoked on the Workspace Subsystem, it sends the Change Package to the Repository Subsystem to request the creation of a new revision. In case that the version of project data model on the client node is older than the version on the server node, the Repository Subsystem rejects the create revision operation and the Workspace Subsystem requests the user to update his project data model before committing. In case the commit succeeds, the Repository Subsystem on the server node acknowledge the change, and sends the Change Package to all online client nodes, which are registered for push updates. The affected online clients proceed as in the online mode. Figure 5.6 shows a simplified sequence diagram of the control flow in offline mode.

5.7 Boundary conditions

The boundary conditions describes the system configuration, startup and shutdown of subsystems and the handling of exceptional conditions [19, p. 277]. This section explains how the Sysiphus is configured, initialized and shut down and how exceptional situations are handled.

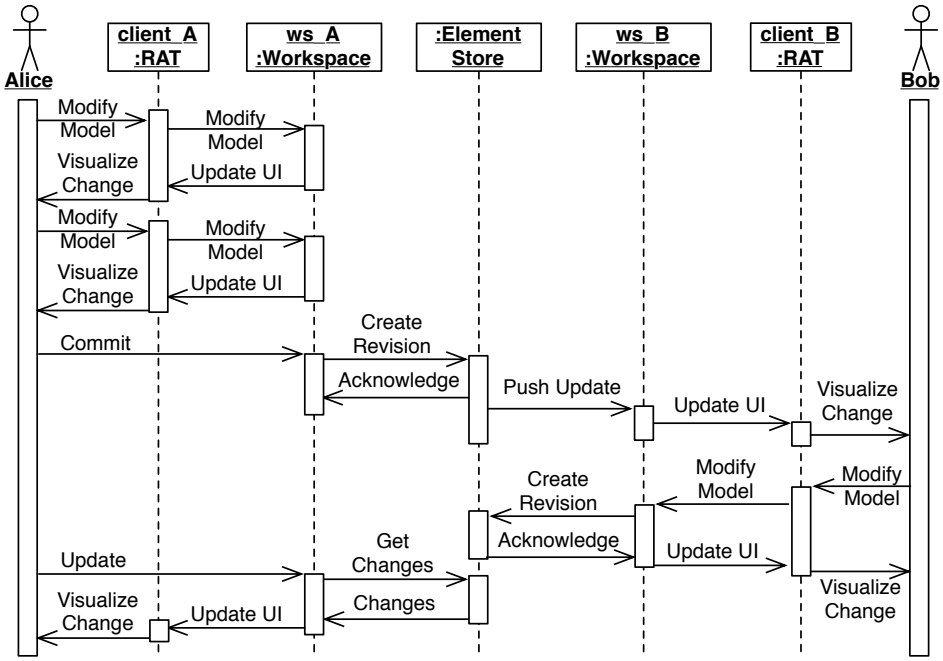


Figure 5.6: Control flow in offline mode (UML sequence diagram)

The sequence diagram shows the two end-users Alice and Bob, both working with RAT on distributed hardware nodes. The RAT instance of Bob is constantly connected with the Element Store, and is registered for push updates on changes. Alice works offline and only connects to the Element Store server for commits and updates. Alice uses her client application to modify the model several times, before she commits her changes. The Workspace Subsystem connects and sends all changes gathered so far to the Repository Subsystem and creates a new revision. The repository sends a push update with the changes to the online Workspace Subsystem of the client application of Bob. Bob, working in online mode, creates a new revision by modifying the model. Alice will only receive his changes when she manually updates her workspace.

5.7.1 Configuration

We separate the configuration description between the Element Store server component and all client applications. The Element Store server has a textual configuration file that defines all used network ports, and contains the required information for the Storage Management Subsystem. It defines if the data is stored directly in files, or if a database management system is used. In case of the file-based approach, the system file path to the store location must be provided. Otherwise, the required database information like the database host and port, as well as the user name and password used to access the database must be provided.

Client applications like RAT needs to know the Element Store server information, like the host name and used server port. The information of several servers may be provided at the login and the user can select the server he wants to work with. The login user interface component provides the additional option to manually add the description of a new server. The server information data gets stored in the user home directory and can be reused by any Sysiphus client applications.

5.7.2 Startup and Shutdown

Startup and shutdown need to be considered for the client applications and the Element Store server of the Sysiphus system. When starting the Element Store server, the Repository Subsystem loads all stored project data models and the version object models by using the Storage Management Subsystem. The version object model of a project may contain many versions and many changes, which require much memory. The memory usage can be reduced by constraining the loading to a configurable depth, starting from the head revision. The remaining history is loaded on demand only.

When starting a client application in offline mode, the Workspace Subsystem loads a requested project, which has been previously checked out, as well as any uncommitted changes by using the Storage Management Subsystem. When working in the online mode, the Workspace Subsystem loads a copy of the requested project from the Element Store server.

On shutdown, the Repository Subsystem on the server node notifies all registered online clients about the shutdown. This enables the client application to perform a clean shutdown as well. Shutting down a client application in online mode requires to close all it's workspace projects and to unregister from the Repository Subsystem on the server node.

5.7.3 Exception Handling

To satisfy the reliability design goal, it is especially important for all components of the system to react adequately to exceptions. In case of an unexpected or unrecoverable exception in the Element Store server, the Repository Subsystem

triggers an emergency shutdown to avoid persistent data corruption. The Storage Management Subsystem is always notified after all changes are applied to the project data model and the version model. This ensures that data is only stored when it has been successfully processed by all the other involved subsystems.

The Workspace Subsystem provides a rollback operation, enabling Model Layer subsystems to rollback Model Operations that were interrupted by an exception. When an exception occurs while a Model Layer subsystem is performing a Model Operation, it shall invoke the rollback service operation of the Workspace Subsystem. The Workspace Subsystem removes the unfinished and therefore possibly inconsistent Model Operation from the Change Package, holding the uncommitted changes. All executed changes of that operation are reverted.

Network failures on client nodes are handled differently, depending on the mode of the Workspace Subsystem. Client applications that are working in the online mode are closed immediately, when the connection to the repository server is lost. The loss of the connection is an unrecoverable situation, since the online synchronization mechanism requires and relies on an uninterrupted connection to the Element Store server repository. Offline workspaces only require a connection to the server repository when operations on the repository are explicitly requested by the users. Consequently, the users can be notified about connection failures and are asked to retry, when the network connection is established again. The client applications must not quit and the users can continue to work offline.

CHAPTER 6

Applications and evaluation

The goal of this chapter is to show the validity of the concepts introduced in this dissertation. Since 2000, we have used and evolved successive versions of Sysiphus. Sysiphus has reached a state of maturity that it has been used in industrial and academic applications. In academia, Sysiphus was used in software engineering project courses, for teaching software engineering concepts, for evaluating new research concepts, and to manage system development projects from the chair of Applied Software Engineering and from interested students.

Our software engineering project courses provide a realistic environment to students who experience first hand the complexity of software development projects. These projects feature an actual problem specified by a real client and sometimes involve up to 100 participants in several locations, in Germany, in the United States, and in New Zealand. These courses have provided us with a research testbed that is realistic enough for quickly evaluating ideas while enabling detailed observation and surveys [20, 39]. The project courses are a first approximation for a distributed project, as students work part time, from different locations, and usually do not know each other before the start of the course. Moreover, students elicit requirements and deliver a system to an actual client external to the university.

In industry, we used Sysiphus in consulting projects. After taking into account the feedback from the academic applications, we have used documents and feedback from industrial partners to assess how realistic our ideas would be in real projects. This enables us to study aspects related to scale and long term use that we cannot assess in an academic environment.

Section 6.1 describes the projects that were used to evaluate the developed concepts and the realization of the requirements provided in chapter 2. In section 6.2 we provides additional applications of Sysiphus that were used to support teaching software engineering concepts.

6.1 Case studies

This section provides anecdotal evidence on selected case study projects. The projects were conduced simultaneously to development of the introduced concepts. The results of a postmortem analysis from each project were used to iterate and incrementally refine on the concepts. We describe project environment and the usage of Sysiphus for each case study. By observing the projects, analyzing the created RUSE models in Sysiphus, and conducting interviews with the project participants, we investigated the projects for four goals:

- The feasibility demonstrations of the introduced concepts in a wide range of projects.
- The evaluation and refinement of the requirements introduced in chapter 2.
- The feasibility of communication and capturing rationale within the system modeling context for the whole life cycle of a project.
- Investigating the concept application differences between beginners and experts.

		Requirements								
Year	Project	Artifact integration	Consistency of multiple model views	Support for fine-grained search and filter mechanisms	Configurable labeling and categorization	Accountability and access control	Traceability	Awareness	Configuration management	Extendability
2003/ 2004	Cargo & Logistic	X	X							
2004/ 2005	CampusTV	X	X							
2005	Mobile Sportsman Artifacts	X	X			X	X			
2004/ 2005	Symphonia	X	X	X			X			X
2005/ 2006	Virtual Symphony Orchestra	X	X	X	X	X	X	X		
2006	IBM Awareness Mockup							X		X
2006	JASS									X
2004- 2006	MOQARE	X	X							X
2007	Yieeha	X	X	X	X	X	X	X	X	
2007	TEAM	X	X	X	X	X	X	X	X	

Figure 6.1: Requirements evaluation

The table provides an overview of the projects of this sections and describe which requirements are evaluated.

6.1.1 Cargo & Logistics

During the winter semester 2003/04 the project Cargo & Logistics [22] developed a functional prototype forecasting system for the chemical company WACKER Chemie AG [125]. The system analyzes the logistic information provided by the SAP system of WACKER and calculates and visualizes a forecast about the logistics of inventory management, freight service support and dispatch disposition (see figure 6.2). The project involved 30 students that worked in four teams.

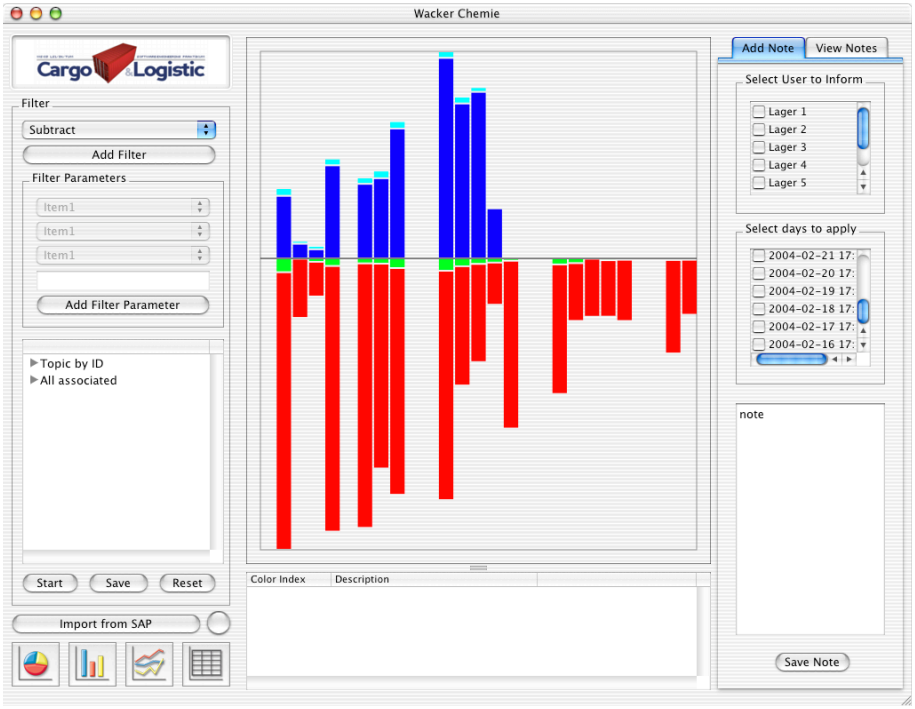


Figure 6.2: The functional prototype of the Cargo & Logistic project course
The figure shows the functional prototype, visualizing and forecasting the logistic information of the company WACKER Chemie AG.

In this project, Sysiphus was used the first time for an integrated development of the requirements and the analysis models, including functional and nonfunctional requirements, actors, use cases, and analysis class models. The requirements analysis document was generated with Sysiphus. At that time, Issues were already supported in Sysiphus, but Comments and Work Items were not. The system design was created with several tools and documented in Word. We used Lotus Notes-based BBoards (Bulletin Boards) as a communication platform for threaded discussions and for capturing Issues. Moreover, the BBoards were used to create project wide announcements about the activities and tasks the participants had to perform. The organizational model was provided by a web-based address book realized with Lotus Notes.

In this project initial parts of the *artifact integration* and the *consistency of multiple model views* requirements were realized and resulted in a generated, sound, and consistent requirements analysis document, which had a higher quality than the documents of previous projects. The requirements analysis document turned out to be one of the primary client deliverables, as it was used by the WACKER IT department for realizing a production version of the system.

However, we recognized that the usage of different tools led to problems among the students and the delivered work products. The system design document that was created with other tools than Sysiphus was inconsistent in itself, and in relation to the requirements analysis document. We were not able to trace from the requirements to the subsystems and the components of the system design and could not identify if all requirements were addressed, and which requirements were realized by which subsystems. In a postmortem analysis we were not able to access the rationale behind the analysis or the design. The students did not use Sysiphus to create and capture the rationale in form of Issues, Proposals, and their Resolutions. We recognized that most Issues were raised during informal communication in Lotus Notes. Students discussed about problems before they were identified as relevant Issues. Therefore, the Issues were represented in detail in informal communication threads in Lotus Notes, but were not explicitly created in Sysiphus. Due to the project's time pressure and missing responsible participants, the Issues were not recreated in Sysiphus. As the informal communication threads from Lotus Notes were not connected to the system model elements in Sysiphus, it was not feasible to identify the rationale and related collaboration artifacts for the system models.

The project showed that the traceable integration of the requirements and analysis and generation of the requirements analysis document leads to consistent models and documents, which we did not found in previous project courses. Therefore, we decided to follow this approach by extending the RUSE model with system design artifacts. We recognized that the separation of collaboration and system modeling hinders the efficient capturing of rationale. The students did not switch their context and tool when recognizing that their discussion leads to an Issue. We decided to extend the RUSE model with Comments to support informal communication and discussion threads in the context of system modeling.

6.1.2 CampusTV

During the winter semester 2004/05 the project course CampusTV [23] developed a prototype system for supporting interactive and distributed lectures, based on digital video broadcast (DVB-T). The hardware transmitters were provided by the company and our client Rohde & Schwarz [108]. The developed system records and transmits a digital video from the lecturer and the lecture slides in realtime over DVB-T. Clients can view the video from distributed locations and can interact by positing questions. All questions are collected and available to all

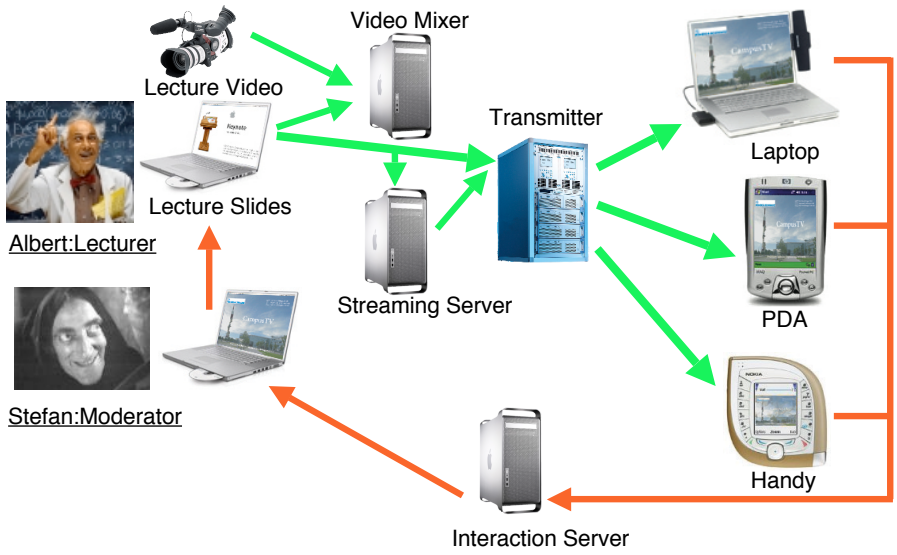


Figure 6.3: Visionary mockup of CampusTV

The lecturer and his slides get recorded and transmitted over DVB-T in realtime. Mobile devices receive the video signal location independent have the possibility to interact by posting questions. Questions are tracked by a moderator and selectively forwarded to the lecturer.

other clients, which are also able to answer them. A moderator tracks all questions and forwards questions with class wide importance to the lecturer, who can discuss the question openly (see figure 6.3). The project involved 25 students that worked in four teams and selected students worked in additional cross-functional teams.

The project used Sysiphus to develop the requirements analysis and the system design documents, including the related Model Elements. Moreover, Sysiphus supported Comments for informal communication. Unfortunately, Sysiphus had no awareness support and did not notify participants about changes and ongoing collaboration. Email notifications about posts to the BBoards were supported and so we decided to additionally use Lotus Notes. The assumption was that Sysiphus is used to communicate about the system models and capture arising Issues, while Lotus Notes is used for the project organization and the notification of current tasks.

The RUSE model of the project shows that the assumption was wrong. The students created only one Comment and one Issue in Sysiphus. Instead, the BBoards were full of discussions about the system. We had the same problems in tracing between the system models and their Issues and the related communication than in the Cargo & Logistic project. It was not feasible to identify the rationale behind the developed system. Although we conducted rationale tutorials at the beginning of the project, interviews showed that the students assessed the rationale management as the most unimportant task of the project. Most students were beginners in

system modeling and programming, so that the system development was already challenging under time pressure. In their opinion, the capturing of rationale was the first task to skip to meet the deadlines.

Satisfying was the integration of the system design into the RUSE model. The students created 269 Model Elements from requirements, analysis and the system design and generated the related documents. Traceability links between the related Model Elements were created and were used to maintain consistency across the system models and documents.

6.1.3 Mobile Sportsman Artifacts

The development project Mobile Sportsman Artifacts (MSA) involved four students and was an interdisciplinary project between the Information Systems and Preemptive and Rehabilitative Sports Medicine chairs of the Technische Universität München. The six month project started in January 2005 and developed a system that manages personalized artifacts, including training and health relevant information for long-distance runners. The system captures semi-automatically health information like the pulse or blood pressure from relevant sensors and calculates goal oriented training recommendations. The personalized artifacts are accessible by web-browsers or over mobile hand held devices (see figure 6.4).

Figure 6.4: The web interface and the mobile interface of the MSA system.
The left part of the figure shows the HTML-based web interface of the MSA system. The hand held device on the right shows mobile interface of a training schedule for a long distance runner.

The students were excellent programmers and had already deep knowledge of system modeling. They started system modeling with Rational Rose [62]. But after initial usage, they decided to switch to Sysiphus. They claimed a missing support for distributed collaboration of Rational Rose and wanted to capture rationale knowledge in form of Issues. We provided Sysiphus in the same version that was used in the CampusTV project. They successfully modeled the system with Sysiphus, including the requirements, the analysis, and the system design.

The generated requirements analysis and system design documents were part of the project deliverables.

The issue model was intensively used to capture and discuss problems about the unknown problem domain, as well as about the design. They created 129 system model elements and 18 Issues. We use the ratio between Issues and system model elements to measure for the reusability of the developed system models and call the ratio issue coverage. Calculating the issue coverage shows that 13.95% of the system model elements were discussed with our issue model. In contrast, the issue coverage in the CampusTV project was 0.37%. We explored the reasons for the big difference by interviewing the participants. We figured out that all participants had previously been involved in development projects and some had already experienced the draw backs of loosing the rationale for critical development decisions. Moreover, they saw a big advantage of investigating the solution space of critical problems in the defined structure that is supported by our issue model.

The success of the MSA project led to a follow up project with new student participants. They had to extend the developed system and were required to reuse and extend the existing system models. The participants reported in interviews that the captured rationale represented by our issue model was extremely helpful to understand the system and the decisions that were made during the first project. As they extended the system design, they were able to trace back to the requirements and their related issues. They investigated the proposed solutions and understood the reasons that led to the resolutions. For still unresolved question, they were able to trace to the related participants of the previous project and contacted them directly. This application initially evidenced the realization of the *traceability* and *accountability* requirement of chapter 2.

6.1.4 Symphonia

To assess the *traceability* and *extendability* of our model in an actual industrial project, we used a set of documents from a Siemens project developing a communication framework for enterprise phone applications. At the time, the project had been running for about nine months, included about 40 people, and was about to be distributed to four different sites, growing to about 100 participants. We selected about 120 use cases and 100 nonfunctional requirements from their requirements documents, and entered in Sysiphus traces for these requirements, including links to design components. We extended our model with project specific system test cases and added traceability links to other related models. The documents we examined also included annotations which we also entered in Sysiphus. We then used the entered models as examples to elicit feedback during interviews with key individuals, then with project management, and finally with all relevant persons as a group. Several participants interviewed had taken part in distributed projects and had used DOORS [121] as a requirements management tool so far.

Within this study, we observed several lessons learned:

- The usage of Sysiphus enabled to visualize and manage the traces from requirements to nonfunctional requirements, design components and test cases within one view. Change impact could be identified through all models, that were previously distributed in several tools and only manually connected by listing relations in external spread sheets.

- Different roles are involved in creating and using the traceability links. For example, architect or senior developers are typically responsible for linking requirements to components, whereas product managers and requirements engineers need change impact knowledge when prioritizing requirements. When the individuals involved are in different sites, we think that the likelihood that traceability links are entered is much reduced.

- Individuals entering requirements in the tool are not necessarily the stakeholders who originated the requirements, as pointed out by Gotel [48]. However, architectural and development decisions seem more easily traceable to their authors.

- The project we studied was organized in four week iterations, at the end of which a product increment, including both documentation and code, was validated. We think that such fast pace iterations are necessary to keep the model up-to-date so that different sites use annotations on the model to collaborate. Conversely, we think that the value of traceability through annotations would be reduced in projects were documentation either precedes the product or is reconstructed after the product is stable.

6.1.5 Virtual Symphony Orchestra

During the winter semester 2005/2006 the project course Virtual Symphony Orchestra (VSO) [127] developed a system that enables children to conduct a virtual symphony orchestra. A video stream of each musician is displayed on one of many screens surrounding the child conductor. The child hears the symphony through a surround-sound system, providing a three-dimensional sound image of the positions of each musician. A camera tracks the baton of the conducting child, enabling the child to interact with the orchestra. The child can change the tempo of the music and volume with motions of the baton. The child can also walk through the orchestra, as the video and audio streams are dynamically adjusted to reflect the relative positions of the musicians and the child (see figure 6.5).

The project started on October 30, 2005 and ended with the Client Acceptance Test on February 16, 2006. The client is the Bavarian Symphony Orchestra [119] with the chief conductor Mariss Jansons [81]. 24 students participated on the project and worked in teams, each lead by a student team leader. Each development team was responsible for a component. A cross-functional architecture team included representatives from each development team. All teams met weekly for

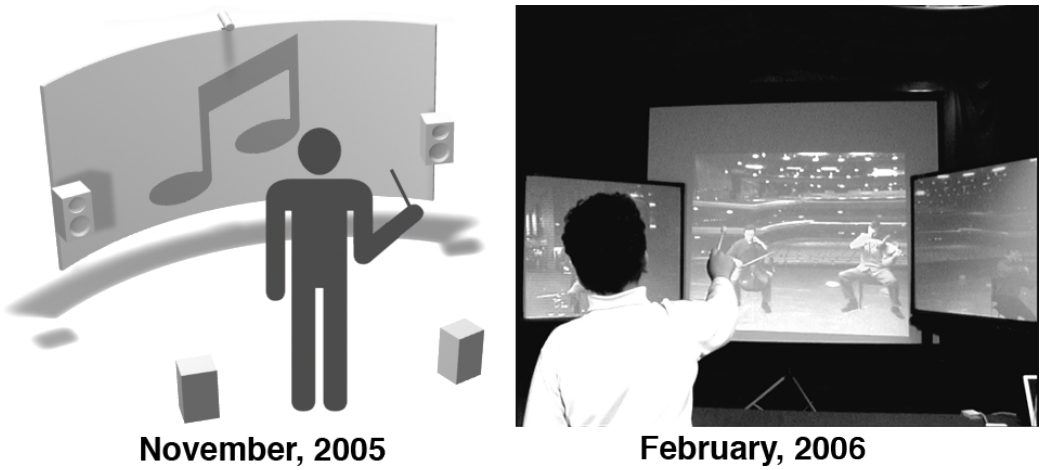


Figure 6.5: VSO system mockup and a picture of the CAT.
The figure on the left shows the visionary mockup of the VSO system from November 2005. The picture on the right site shows the VSO system during the client acceptance text in February 2006.

team status review and issue discussion. The team leaders met with the project manager weekly for project status. A weekly all-hands meeting was used for tutorials and major reviews. As all participants were students, they had no common office, worked in distributed locations, and often met only at team and all-hands meetings.

In difference to the previous project development courses, we did not use Lotus Notes any more. We added the organizational model, the Work Items and the described awareness mechanisms to notify the participants of related changes. Sysiphus was used to develop the requirements analysis document, the system design document, as well as the software project management plan. Comments and the issue model were used to communicate and to capture rationale. The Work Items were used to assign tasks to participants and for the project management activities, like tracking state and progress. The realization of the *support for fine-grained search and filter mechanisms* requirement was used to filter all Comments, Issues, and Work Items to their related Teams. The filter results were integrated into Team related sites of the web-based project portal and showed always the ongoing communication and open Issues and Work Items.

The students created 234 system model elements, 48 Issues, and 141 Work Items. Calculating the issue coverage results in 20.51%, which is much higher than in the previous CampusTV project course (0.37%) and in the MSA project (13.95%). We expected a higher value than in CampusTV, as we completely integrated the project collaboration into Sysiphus. However, we were surprised that the value is higher than in the issue coverage from the MSA project, as the students

of the VSO project were in average less skilled than the students from MSA. The skill levels of the students were equivalent to the CampusTV students. To clarify the result, we explored the captured Issues in detail and recognized that 20 of the 48 Issues were not used to describe and capture system related problems. The students used 20 Issues to ask and resolve organizational questions like "Who will present the actors and use cases?". Removing the 20 Issues from the calculation still results in 7.69% issue coverage.

Additionally, we captured the complete history of accesses during the project. We collected 29930 change events, denoting changes to the Model Elements of the repository and 29151 user events, denoting read accesses and user actions in the client applications (e.g., changes of focus in the traceability graph window). Table 6.1 shows for selected Model Element classes the average number users and teams who read or changed an element of that class. For example, on average, each Work Item was read by 5.76 and modified by 1.78 users. Similarly, on average, each Work Item was read by members of 4.5 teams. The average number of read and write accesses by users show how the system and collaboration models were used to interact and how system knowledge is shared across participants and teams. These users gained knowledge about the focused element and transfered the knowledge by face-to-face communication to their teams. Therefore, the average team access indicates how many different teams were involved and aware of the elements. As some users participated in more than one team, the average number of accesses by teams can be higher than the number of users.

Element Count	Element Type	User Read	User Write	Teams Read	Team Write
141	Work Items	5.76	1.78	4.50	1.98
48	Issues	6.45	2.20	5.81	3.00
60	Proposals	3.95	1,15	3.93	2.00
32	Resolutions	0.75	1.03	0.84	2.03
34	Comments	4.55	1.05	4.11	1.64
13	Scenarios	7.69	2.30	6.46	3.23
24	Use Cases	8.33	4.95	7.04	5.04
23	NFR	7.30	1.91	6.04	3.00
9	Packages	11.66	3.66	7.66	4.00
109	Classes	3.22	1.67	3.27	2.19
10	Sequence D.	8.80	2.10	7.50	2.30
5	Components	9.00	2.80	7.00	3.00

Table 6.1: Average number of unique users and teams reading and writing a Model Element.

We also captured the access date and time and recognized mayor activity peeks (see Figure 6.6) around the Requirement Review, Analysis Review and System Design Review. At all reviews, the students presented the relevant models of the complete system. To ensure consistency and to integrate the different aspects of

all teams, communication, negotiations, and problem solving across teams was required. Combining the data from Table 6.1 and Figure 6.6 suggests that Sysiphus was used to collaborate across sites and teams, while face-to-face meetings were used to resolve team-internal problems. To report problems and to raise clarification questions, students created Issues and attached them to other teams' system models. Examining the content of the Sysiphus repository, we observed that students used our collaboration model mostly for application domain problems, design questions, components API discussions, and for project management aspects. Participants used mailing lists to discuss infrastructure problems, implementation and programming questions. While the evidence from this case study is anecdotal as the subjects were students, it confirms that it is feasible for groups of participants who do not know each other to collaborate using Sysiphus.

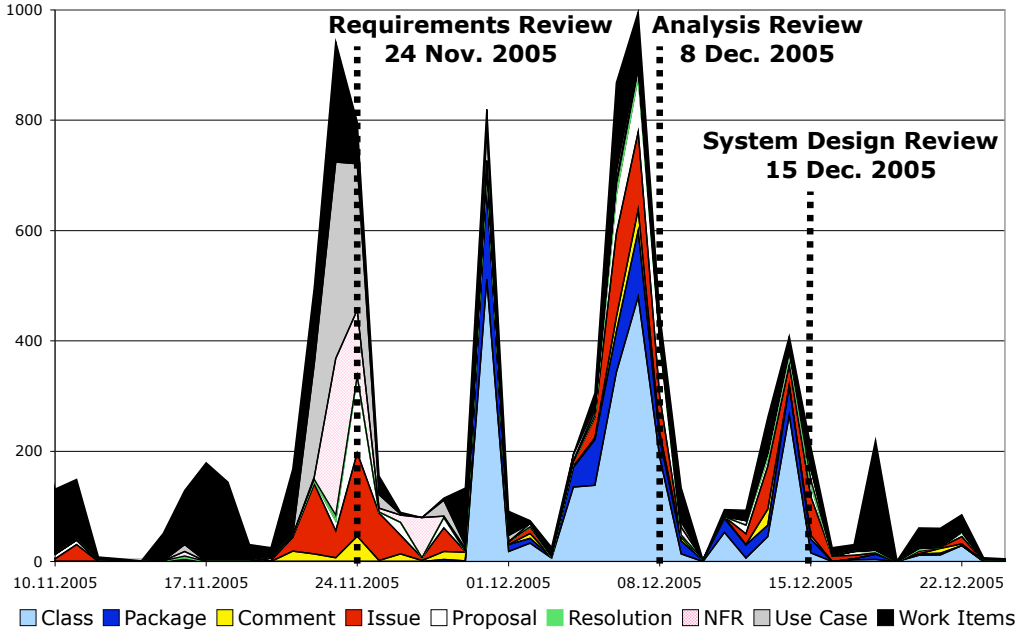


Figure 6.6: User activity on selected Model Elements in the VSO project.

6.1.6 IBM Awareness Mockup

The realization of the *accountability* and the *awareness* requirements was evidenced by Florian Huber [58]. He used the capabilities of Sysiphus to compute and visualize related project participants, based on their assigned Work Items, the system model elements, and Issues. His work was used to mockup new awareness concepts for IBM in Vancouver. We imported the persons and bug reports from the free accessible bug tracking system Bugzilla [86] of the Eclipse [42] project, and combined them with the eclipse components. The Sysiphus client SysClipse was

than used to visualize social networks of Eclipse developers from within Eclipse. Main contributors of specific components and knowledgeable developers for given bugs could be identified (see figure 6.7).

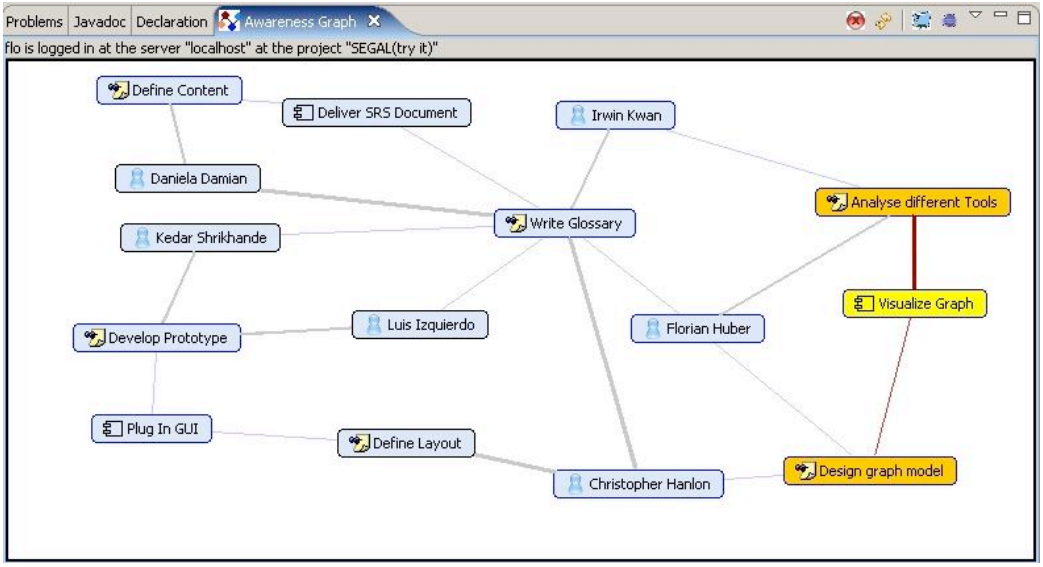


Figure 6.7: Visualized awareness graph in the Sysiphus client SysClipse
The screenshot shows the Sysiphus Eclipse plug-in SysClipse, visualizing relationships between project participants. The relationships are computed on the basis of the participant's activities on Work Items, components and issues.

6.1.7 JASS

The JASS project was part of the Joint Advanced Student School in St. Petersburg in April 2006. We conducted a 10 day project with seven German and seven Russian students. The goal of the project was to extend Sysiphus with an agile meeting management tool for distributed software projects, which is based on our rhetorical issue model and Work Items. Meeting agendas should not explicitly be written by a meeting facilitator, but should be generated automatically from the open Issues and Work Items from the Sysiphus repository.

Beside extending Sysiphus with a meeting manager component, we used Sysiphus to manage the project, following the Scrum [113] management method. We used the Work Items as backlog items and applied the filtering mechanism of the RUSE model Documents and Leaf Sections to realize the product backlog and sprint backlogs. The students managed to develop a functional prototype of a meeting management extension for Sysiphus, while the RUSE model proved to be flexible enough to apply agile management methods.

In a follow up project, Jennifer Schiller [112] extended Sysiphus to support the Scrum management method. The Scrum concepts were integrated and merged into the RUSE model and a graphical user interface supporting the Scrum activities was developed as a RAT plug-in.

6.1.8 MOQARE

The Software Engineering Group at University of Heidelberg [116] conducted several project courses that used Sysiphus to develop and document the requirements, the analysis, the architecture and the design. In addition, the goals of the project courses were to extend Sysiphus with new functionality. For example, in the winter semester 2004/2005, a filter and sort mechanism was designed and developed for the web-based interface application REQuest. In the winter semester 2005/2006, the RUSE model was extended to support the creation and documentation of test cases and detailed test artifacts.

Within two bachelor theses, Sysiphus was extended with the concepts of misuse-oriented quality requirements engineering (MOQARE) [56] and the synchronization of object design and source code. The concepts were designed and specified by using Sysiphus and functional prototypes were implemented. The extendability of the RUSE meta-model and the reuse of the existing client applications enabled a fast implementation and validation in real projects.

Sysiphus was used for accomplishing industrial case studies. In the winter 2004/2005 they conducted a Uveitis case study [55], evaluating the task- and object-oriented requirements engineering approach with Sysiphus. From the lessons learned, they extended Sysiphus with the MOQARE concepts and evaluated the approach in a confidential case study with Porsche. Based on a 160-side client specification and the related 240-side technical specification of a real project, they created the nonfunctional requirement by using Sysiphus and evaluated the project consistency, as well as their MOQARE approach.

6.1.9 Yieeha

The company Yieeha Ltd. & Co KG [132] are developing the Web 2.0 application Yieeha, a playful community-based platform to discover new products. Yieeha users have the opportunity to add and publish discovered products to the platform, as well as to their personal wish list. Personalized product wish lists and profiles arise, which are discoverable by Yieeha users and companies. Online shops have the opportunity to advertise their products by raffling them on the Yieeha platform. Figure 6.8 shows the list of top wishes from the Yieeha users.

Yieeha Ltd. & Co KG recently started to use Sysiphus, after we realized the *configuration management* requirement. They are going to model and manage their requirements, analysis and system design. Before, they specified their requirements and goals in traditional documents. They noticed that the documents

Figure 6.8: The top product wishes of the Yieeha platform
The screenshot shows the top four product wishes of the Yieeha Web 2.0 platform in April 2007.

outdated very fast and that they were not able to manage consistency and traceability across different documents. They expect to overcome these problems by using Sysiphus and will provide us practical experiences and feedback on our approach.

6.1.10 TEAM

In September 2006, the European Commission signed an agreement to fund the 30 month research project TEAM [120], which aims to develop an open-source software system, seamlessly integrated in a software development environment for enabling decentralized, personalized and context-aware knowledge sharing. In particular, it focuses on a semantic-based framework for sharing knowledge about software implementation, accessible from an IDE (see figure 6.9). The project has ten participating organizations from eight different European countries, including universities as well as industrial companies.

The project just started using Sysiphus. The participants initially use Sysiphus to analyze and model their problem domain by using class models. All models are shared across the eight countries in realtime and Sysiphus provides the mechanisms for synchronous collaboration. We hope to get detailed feedback on using Sysiphus across several countries.

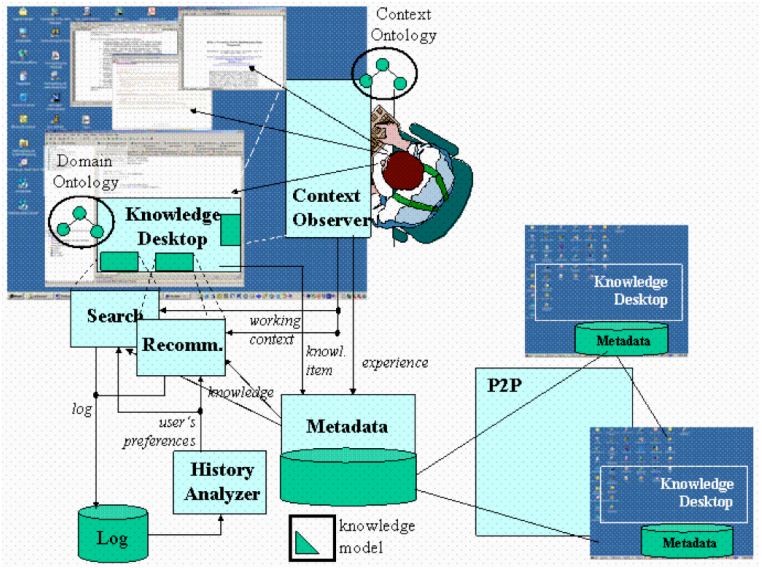


Figure 6.9: A conceptual overview of the system developed by TEAM
The figure shows a conceptual architectural overview of the TEAM system. Developers access and share knowledge by using ontologies and taken their local working context into account.

6.2 Teaching software engineering

We also used Sysiphus for teaching software engineering concepts including rationale management and to develop example projects used in software engineering lectures. This section describes applications of Sysiphus in the context of teaching software engineering.

6.2.1 Arena

ARENA [5] is a distributed, multi-user system for organizing and conducting tournaments. It is an open source project, published under the GNU GPL [44] license. ARENA is game independent in the sense that organizers can adapt a new game to the ARENA game interface, upload it to the ARENA server, and announce and conduct tournaments with players and spectators located anywhere on the Internet. Organizers can also define new tournament styles, describing how players are mapped to a set of matches and how to compute an overall ranking of players by adding up their victories and losses (see figure 6.10).

ARENA has been developed as a companion example for the book Object-Oriented Software Engineering [19]. The goal is to provide a non-trivial and living example for software engineering education. With ARENA, an instructor can cover technical topics (e.g., access control, concurrency control, dynamic class loading), and methodological topics (e.g applying design patterns, specify-

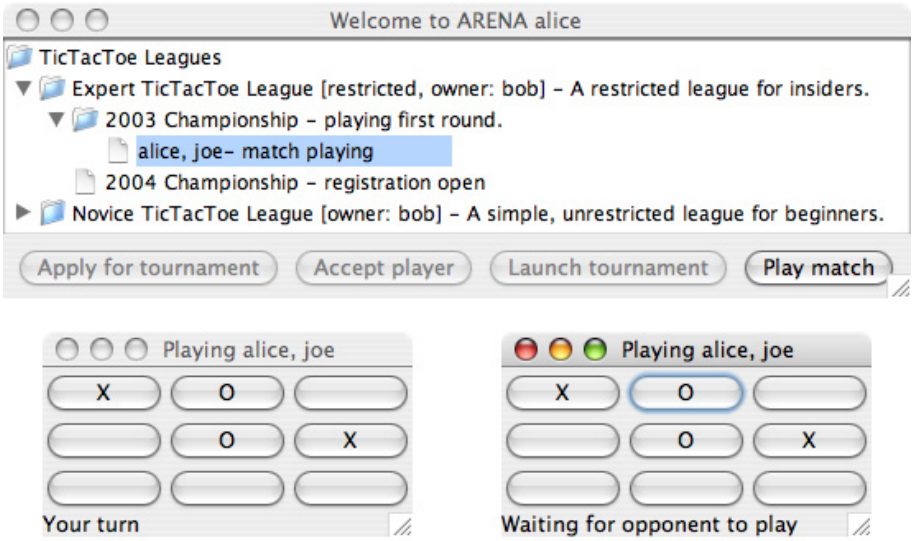


Figure 6.10: Screenshots of the ARENA

The screenshot show the ARENA management interface to organize tournaments and the user interface provided distributed sample game Tic Tac Toe.

ing contracts). ARENA can also be used for supporting project courses in which students extend or refine the system.

We used Sysiphus to create the requirements, the analysis and the system design of ARENA. The rationale of design decisions are captured by using our issue model. All documents and models are frequently published to the ARENA website and provide the documentation for the project. ARENA was used in several lectures. Students had to extend ARENA for new game types. They used the documents and models from Sysiphus to learn about the ARENA requirements and the system design. Interviews showed that the issue model annotated on the system model elements was extremely helpful for their understanding.

6.2.2 Asteroids

During teaching software engineering including design patterns, we recognized that students have problems in understanding and applying design patterns. We recognized that practical experiences are needed to understand and internalize the application of design pattern. Therefore, we developed a set of design pattern exercises that are based on our own implementation of the Asteroids game. Different versions of Asteroids are available, each a target for applying a design pattern. The Students have to realize new requirements by applying a design pattern in the system model, as well as in the source code of Asteroids. Figure 6.11 shows the game Asteroids.

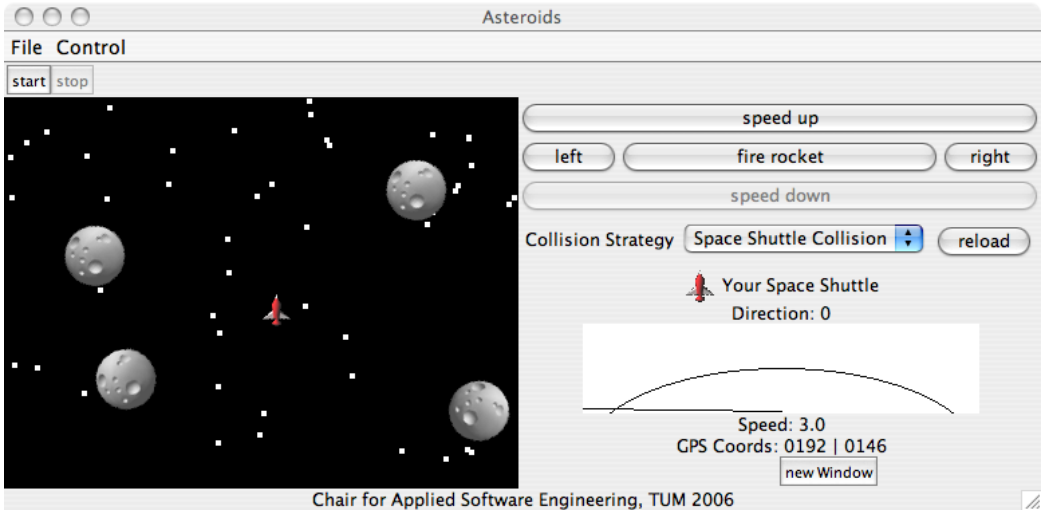


Figure 6.11: Screenshot of the game Asteroids

We used Sysiphus create and manage the requirements and analysis of Asteroids. We created multiple projects, each referring to a Asteroids version, which is target for an exercise. The students used Sysiphus access the models of Asteroids and to apply the design pattern by changing the models.

6.2.3 Software engineering lectures

The Software Engineering Group at University of Heidelberg [116] used Sysiphus to support the software engineering lectures "Planning and execution of software development projects", "Requirements engineering and project management" and "Software engineering for modern applications". The results of using Sysiphus in teaching software engineering were published in [16, 17, 100].

Planning and execution of software development projects
In the summer semester 2004, The University of Heidelberg used Sysiphus to create and document software engineering models. The students accessed the RUSE model of a real project, to learn how they are used. After understanding the concepts, such as requirements, use cases and actors, the students used Sysiphus to change and extend the existing project models. The course was repeated in the summer semesters 2005 and 2006.

Requirements engineering and project management
In the summer semester 2005, students used Sysiphus to get practical experiences

in the area of project management. The students created project management plans, including work-packages, milestones and measured the project states.

Software engineering for modern applications
In the winter semesters 2005/2006 and 2006/2007, Sysiphus was used by the students to get experiences in the area of rationale-based architecture modeling and modern technologies. They used Sysiphus to create a requirements specification, the system design and the test specification. They used intensively our issue model to capture the rationale behind decisions. The teaching approach was published in [109].

CHAPTER 7

Related Work and Previous Research

We separate the related work into research and commercial systems.

7.1 Research

Integrating related software engineering models and the use of rationale, traceability, and awareness for supporting distributed development is not new. Research efforts from the software engineering and computer-supported collaborative work communities have investigated these mechanisms for distributed work to achieve a variety of goals, such as improving distributed negotiation [18], identifying social dependencies [33], using traceability to raise awareness [79], using asynchronous issue discovery and consolidation to reduce the need for synchronous collaboration [30, 74]. While many aspects of these proposals appear similar to our Rationale-based unified software engineering model model, each differs fundamentally either in the goal they achieve or their approach. In this section, we examine how our work complements and extends these proposals.

7.1.1 Rationale and Distributed Work

The goal of much rationale research has focused on capturing decision making knowledge for long term use, contributing to the group memory of the organization. However, such externalized knowledge is also useful in the short term for supporting transfer between sites [37]. For example, EasyWinWin [14, 18], an approach for negotiating requirements, leverages off both rationale and collaborative techniques. Rationale is represented as Win conditions, issues, proposals, and agreements, similar to the QOC [80] model in Sysiphus. During negotiation, stakeholders follow a sequence of stages in which they brainstorm win conditions, de-

fine glossary terms, prioritize Win conditions, surface issues and constraints, and resolve issues. The EasyWinWin tool enables stakeholders to remain anonymous during several stages of the process, thus shifting the focus of the negotiation on the content. The tool enables dealing with a large number of win conditions and with geographical distribution. While EasyWinWin and Sysiphus are similar in their use of rationale, they differ in their scope: EasyWinWin targets high-level requirements elicitation during synchronous workshops, Sysiphus focuses on detailed development of requirements and architecture during both synchronous and asynchronous collaboration. This key difference lead us to investigate closely the relationship of rationale with traceability and awareness.

7.1.2 Traceability and Awareness

Requirements traceability has long been identified as a key success factor in large software development projects [35, 47]. The Standish Group's CHAOS report identified incomplete and changing requirements as two key factors in project failures [117]. Moreover, requirements traceability is required for CMMI compliance [114]. While commercial requirements traceability tools (e.g., DOORS and RequisitePro) have been used in distributed projects, their focus is on managing complex requirements and on change management. They do not support distributed collaboration and traceability to the project participants. The interest of using traceability for supporting the concept of group awareness is recent.

Souza et al. [33] conducted a field study on the use of Application Programmer Interfaces (APIs) for coordinating distributed work. While they confirmed the popular belief that APIs create walls between teams allowing them to work independently, they found that defining contracts with APIs can also create obstacles for inter site collaboration, in particular, during the integration. They also noted difficulties by developers in finding and communicating with their counterparts at remote sites. Based on these insights, Souza et al. propose the concept of *social call graphs*, depicting both dependencies among components and their authors, thus depicting the social dependencies between developers that would be critical during integration. The RUSE model provides pretraceability with a similar graph of relationships between model elements and authorship, extended this concept to software engineering models in general.

ADAMS [79] tracks artifacts produced during development along with their dependencies. Moreover, ADAMS associates developers with the operations they are allowed to perform on each artifact, thus also capturing life cycle aspects. Developers become aware of changes to dependent artifacts by subscribing to events. To ensure that all dependencies are captured, ADAMS uses a latent semantic indexing mechanism to suggest the addition or removal of links between artifacts, based on their linguistic similarity. Traceability and notification are thus used to raise the awareness of developers to changes to related artifacts. Latent semantic indexing ensures that links among artifacts produced at different sites are cap-

tured. ADAMS provides a generic file-based representation of artifacts, but does not explicitly represent software engineering concepts. Conversely, in Sysiphus, users can specify such concepts when subscribing to notifications. As indicated before, a user can request to be notified when high-priority use cases change.

7.1.3 Asynchronous Inspections

Several web-based tools have been proposed for supporting asynchronous inspections in distributed projects (e.g., IBIS [74] and HyperCode [101]). The goal of such tools is to minimize the number of synchronous activities during the inspection process to enable its distributed execution while retaining the benefits of inspections. Asynchronous inspection has also been used for requirements [30]. While such inspection tools focuses specifically on the inspection activity, they share with Sysiphus the concept of raising and resolving issues asynchronously and attaching issues with the artifact under consideration (e.g., requirements, source code). Our approach differs in that it does not support specific stages in the process, treating development as a continuous process in which issues can be raised, discussed, and closed in any order.

Ebert and Neve summarize in Surviving Global Software Development [40] experiences and best practices for global distributed software development projects. The results focus on project organization structures, process management and integrated workflow management. They stated out that one of the real challenges is to spread the awareness, communication and knowledge to all development levels and sites in real time, specially when different cultures and languages are involved. Within this paper, we proposed a model and prototype implementation to increase these properties by realtime collaboration on requirements and system models integrated with rationale knowledge and communication. Traceability between requirements, system models and rationale knowledge and communication is supported to identify change impact as well as critical stakeholders.

Battin, Crocker and Kreidler describe in [6] the main issues and solutions of a global distributed project from Motorola. Main issues they encountered are the *loss of communication richness* between different sites and a centralized *software configuration management*. The solution approach for the *loss of communication richness* includes the exchange of documents and work products over the intranet, using a web site for each site. The RUSE model follows the same principle, as all artifacts like documents, models, rationale and communication elements are located in the central repository of Sysiphus, accessible from all sites. Thus, awareness of the other site's work increases and foreign experts can be found.

Chisan and Damian developing a model of awareness support in software development in GSD [24]. They propose a workspace, containing all development artifacts like the requirements or the design document, which support notification to relevant project participants, when the content of the workspace changes. To avoid broad cast notification, the model proposes artifact dependencies to notify

selected people only. We agree to the need of awareness in distributed software development and have already implemented aspects of the proposed model. Our approach consists of a central repository containing all development artifacts that are an extension of the described meta-model. Our concrete development artifacts range from documents (e.g., requirements, design) to UML models like use cases, classes, or components to design rationale based on QOC [80]. Awareness is supported as developers can subscribe to their project and get notified when elements changes. Instead of getting notified about all changes they can define the artifact class or the document sections they want to be aware of. Notification of relevant artifacts is supported by tracing to the collaboration model. Authors of system model elements get notified when collaboration artifacts like comments, issues, or work items are attached to the elements.

Mockus and Herbsleb show in [85] the need for finding experts in distributed projects. They introduced a tool that supports the measurement of expertise and expert searching in terms of people or organizational units. The tool works on the data of version control systems (VCS), capturing deltas of files. In addition, author information, modification dates and a change log are captured. The tool enables the browsing and searching of work products and expertise-based identification of people. The work products are mainly source code files and are visualized as a product hierarchy resulting from the source code directory structure. VCSs are very good in dealing with text files, but they have problems when working on binary files, which are mainly used by CASE tools to store models. Often, MS Word files are used to create the project documents, containing images of diagrams. The integration of the configuration management into the RUSE meta-model captures similar change data as a VCS. Therefore, the same approach as in [85] can used to identify experts within the requirements, system models, rationale knowledge, or within the communication elements. The introduced traceability concepts further helps to identify dependencies among experts, expertise, and work products in terms of model elements.

Dellen, Kohler and Maurer propose in [34] methods and techniques to extract automatically *causal dependencies between decisions from development tasks decompositions and from the information flow of fine-grained software process models*. The decisions are part of rationale information and supported with cons and pros. Decisions are made during tasks, which are part of the process model. The decisions of a task are influencing depending tasks. The application of their approach overlaps to ours. The dependencies are used to identify change impact, when a decision changes or becomes invalid. In opposite to our work, they focus on a process model and use development artifacts as inputs and outputs for tasks. We focus on all kinds of development artifacts and support direct links between them and rationale, capturing the reasoning behind them and providing a context to resolve issues.

REMAP [103] is a tool to support stakeholders to create design solutions from a set of requirements, using *Design Rationale*. The rationale is used to capture

knowledge and to negotiate over open issues, related to requirements. A decision leads to a constraint, which must be addressed by the design solution. Therefore, REMAP uses rationale to maintain traceability between requirements and design solutions. The use of rationale in REMAP is similar to ours. In addition, we supports a mechanism to create traceability links. We allow to attach issues to all elements of all software development activities and we integrate rationale into the graphical modeling user interface. Integrating rationale in the developer's daily work tools, increases the acceptance, usage, and benefit of rationale.

7.1.4 Versioning of software engineering models.

Several research efforts addressed the problem of designing SCM systems that are capable of managing complex data structures in a fine grained manner. Most of them focus on single project artifacts such as diagrams rather than consistently managing the complete set of artifacts produced during the life cycle of a software development project.

One of the central techniques described in this dissertation is the representation of deltas using the original editor operations. This idea was heavily influenced by the work on operation-based merging by Lippe and van Oosterom presented in [78]. They discuss the advantages and disadvantages of using operation-based delta representations to capture the semantic context for diffing, merging and several algorithms for conflict detection.

Rho and Wu describe an approach for configuration management of software diagrams that they use in the DIVERS system in [105]. They use a fine grained model with operation-based deltas similar to our approach. Software diagrams are represented as a graph consisting of nodes and edges. In contrast to our work, they focus on managing single diagrams rather than an integrated model of software engineering artifacts. Consequently, they do not show how their approach can be applied to consistently manage a set of interrelated models that have different domain specific constraints.

Ohst et al. present an approach for versioning software documents with a focus on UML-based analysis and design diagrams [96, 98, 97]. They assume these documents are modeled in a fine grained way using a syntax, which is defined by a meta-model. Differences are visualized using a unified document containing both common and differing parts. Versioning is essentially state-based. While tool and design transactions support the grouping of logical changes similar to our Change Packages, the original tool operations are not captured to provide the full semantic context of changes. Thus, differences between two versions have to be incompletely recalculated. Furthermore, they use a pessimistic locking scheme for collaboration. Finally, their approach again concentrates on single diagrams and the authors do not show how their approach can be applied to sets of interrelated models with different domain specific elements.

Nguyen et al. introduce the Molhado framework for building object-oriented software configuration management services with versioning at adjustable levels of abstraction and granularity [88, 90, 89]. Their motivation to develop this framework is the impedance mismatch between the complex data structures of object-oriented design documents and file-oriented SCM systems. The system is geared towards versioning sets of interrelated model elements, like the approach presented in this dissertation, in order to consistently manage the different artifacts created during the software development life cycle. However, the system operates on a very abstract meta-model level only. Consequently, the issue of semantic integrity of domain specific models is not covered.

Mehra et al. present generic version control, visual differentiation and merging for diagrams in the Pounamu meta-CASE tool [82]. Pounamu allows the user to specify and generate multi-view visual design tools for sets of diagrams based on a model of entities, associations, connectors, shapes and manually specified integrity constraints. The paper describes how the authors added versioning, diffing and merging facilities. Different versions of diagrams are serialized into XML and stored in a regular CVS repository. Differences are extracted by comparing the XML documents and translated back into editing operations for the original editors. This facilitates leveraging the editors' logic for diagram specific visualization of diffing and merging as well as their logic for integrity constraint checking. The approach used is essentially state-based since the operations are recalculated rather than stored directly. Although this facilitates the usage of a main-stream SCM system, it loses information about the changes irrecoverably and makes difference calculation more complex and time consuming.

Oda and Saeki propose a fine grained version model for software engineering models focusing on diagrams in [95]. They argue, that SCM for diagrams is highly dependent on the specific types of the diagrams. To solve this problem, they propose an approach that uses a meta-CASE tool to generate the SCM functionality along with the CASE tools from meta descriptions for specific diagram types. They use a graph structures meta-model that differentiates between logical and notational information to represent different diagram types. In their SCM approach they use change-based versioning with operation-based deltas. The facilities for capturing operations during manipulation of the diagrams are automatically generated by adding appropriate editor and capturing code during tool generation. One disadvantage of this approach is that new editor tools have to be generated each time a diagram type is added or modified, thus severely limiting extensibility when tools have already been deployed. Furthermore, they only show how their approach can be applied to single diagrams and not to an integrated model of software engineering artifacts.

Oliviera et al. present Odyssey-VCS, a system for fine grained version control of UML model elements [99]. A main design goal of the system is to support a wide range of CASE tools. Consequently, the MOF and XMI standards are used for the internal representation of diagrams and only a narrow file or plug-

in based interface to higher level tools is provided. The versioning and conflict detection behavior can be configured for each element type on a per project basis. The version model used is state-based and operates on the meta-model level only. Differences are extracted by comparing the XMI syntax trees. Since Odyssey-VCS does not use operation-based deltas and operates on the meta-model level only, model integrity can not be guaranteed during merging.

7.2 Commercial Systems

The IBM Rational RequisitePro [61] solution is a requirements management tool using familiar document-based methods. IBM Rational RequisitePro relies heavily on Microsoft Office products for editing model elements. Changes can be tracked on a per element level. In contrast to the RUSE model, model elements are textual paragraphs of a document with no complex internal structure and no type consistency. Differences can be derived by text-based differencing. A change and version history is provided but changes can neither be undone nor replayed. Offline operation is only supported with a pessimistic locking mechanism. The integrated SCM functionality thus only provides the logging of changes. For any other required SCM functionality such as version selection, an external SCM tool is required. However, such a solution would not be able to supply meaningful changes since it is not fine grained and adds usability issues by requiring the user to use separate tools. Informal collaboration artifacts and capturing issues for representing rationale are not supported.

Telelogic DOORS [121] is a requirements management tool with focus on textual requirements. Changes are tracked on a per element level and can be aggregated into change notes, an equivalent to our Change Packages. Configuration management, traceability, and an extension for use case modeling is fully supported. In contrast to our approach, collaboration models and the organizational models are not supported within DOORS.

IBM Rational Software Architect (IRSA) [63] is an integrated design and development tool. It leverages model-driven development with integrated UML support for creating the architecture of applications and services. The organizational model and informal collaboration and rationale management of a project are not supported. IRSA is built on the Eclipse IDE and uses the Eclipse Modeling Framework (EMF). Kim Letkeman discusses the approach taken to compare and merge UML models in the IRSA in a multi-part article [75][76][77]. IRSA captures changes on one EMF level while user interaction and integrity checking take place on the higher UML level. It turned out to be very difficult to reconstruct the higher level operations at a later stage and to maintain model integrity without capturing additional context. Our approach provides this additional context with the introduced Model Operations.

Artisan Studio [115] is an integrated suite of UML modeling tools for the development needs of technical systems. It can be extended by a module called *Multi-user Change Tracking* to allow change tracking. Artisan does not support workspace isolation. Artisan Software suggests the use of a mainstream SCM tool for all other SCM functionality except change tracking. As only change tracking is supported in Artisan, there is no way to automatically derive meaningful changes between two versions of an Artisan project. Furthermore, an offline mode is not supported by Artisan.

CHAPTER 8

Conclusion

This chapter summaries our research and the results in section 8.1 and outlines future directions in section 8.2.

8.1 Summary

This dissertation addresses the problems of inconsistencies between related artifacts and inefficient collaboration in distributed software development projects. The artifacts such as documents and models are created and managed in separated tools and meta-models that are insufficiently integrated. The same models have usually several redundant media representations, for instance in a modeling tool and in several documents. Artifact dependency traces across all related artifacts and redundant media representations are not supported. Inconsistencies between related artifacts usually arise when changes occur. The project participants have to analyze the change impact to related artifacts and change them accordingly. In distributed development projects with huge numbers of models and documents, the identification of related depending artifacts and redundant media representation is not feasible due to missing dependency traces. Efficient collaboration between the distributed project participants is needed.

Unfortunately, project collaboration is also hindered by the distribution. In particular informal communication to overcome urgent issues is reduced. Missing cross-site project knowledge and awareness of activities and problems from teams located at foreign sites are problems. The rationale behind the parts of a system developed at one site is usually not propagated to all sites and gets lost in long-term projects.

Our approach to overcome these problems is to integrate all related system models, the project organizational models and the collaboration models in the RUSE model that is based on an extendable meta-model. The model is represented in a single central repository and accessible from distributed sites. De-

pendency traces across all artifacts are created and managed implicitly within the same model and support consistency maintenance of related artifacts. The organizational models consist of participants and teams and their activities of manipulating the RUSE model elements are captured. The relations between participants and related model elements are traceable in both directions. The collaboration models include Comments, an issue model, and Work Items. The comments are used for capturing informal discussion threads. The issue model is based on Question, Options, Criteria (QOC) [80] and facilitates the capturing of rationale and represent project knowledge. Work Items describe the project activities and tasks and can be assigned to the project participants. The collaboration models are attached to relevant system model elements and their relations are traceable in both directions. Specifically, our approach is able to:

- Visualize complex relationships across related artifacts from different development activities by multiple distributed teams.
- Eliminate fragile traces that are intrinsic to the use of different tools and media for storing work products.
- Support the identification of related artifacts for mitigating the problem of consistency maintenance.
- Capture rationale knowledge during the whole software development life cycle and to make the rationale accessible to all distributed participants.
- Support distributed informal communication in the same context as developing the system models.

We realized our concepts in a tool called Sysiphus. We applied Sysiphus in a wide range of different projects to demonstrate the feasibility of our concepts. The projects range from small projects with 4 participants to large projects with about 40 participants. We observed university project courses, independent student development projects, industrial projects, and projects that were used for teaching software engineering concepts. The skill level of the project participants range from beginners to experienced professionals.

Our observations show that the integrated RUSE model and the traceability concept presented in this dissertation can be deployed and are useful in a distributed development environment. We also observe that missing traceability links will probably be most often those between model elements owned by different sites or by different roles. In this case, it is critical that sufficient collaboration (either in the form of informal Comments or formal Issues) occurs over the model to make up for this deficit. Expert project participants applied our collaboration concepts and in particular the capturing of Issues immediately, while beginners had more problems.

In summary, the applications evidenced that our approach is feasible for a wide range of distributed projects. The applications in our university development project courses resulted in system models and documents with better quality than

those, created in previous project courses. The ratio between Issues and system model elements is probably a good measure for the reusability of the developed system models. By iterating and refining our concepts, we increased the ratio from 0.37% to 20.51% in our project development courses. The applications in industrial projects confirmed the need and the usability of our concepts by professionals.

8.2 Future directions

While this dissertation has been a step into the right direction, more independent case studies of long running industrial distributed projects need to be performed to measure the impact of the proposed concepts. Siemens has already done three pilots projects without Sysiphus that incorporated nonfunctional and functional requirements into UML [12]. The pilots were successful and confirmed that we are on the right track. During the requirements engineering effort on large projects, Siemens have experienced significant problems with the use of different media for the storage of related materials. They have run into problems with synchronization and fragility when generating traces using third party tools that store the traces external to both the source and destination. Essentially, they added another storage media to the mix. We are currently planning an industrial case study in cooperation with Brian Berenbach from Siemens Cooperate Research to evaluate our proposed concepts and compare the results with previous project from Siemens.

However, our concepts and the realization of Sysiphus have already reached a level of maturity that new research projects are based on our work.

The research of Korbinian Herrmann focuses on Model Elements that are on different abstraction levels. For instance, a class may arise during the problem analysis and gets refined and detailed during object design and detailed design. Moreover, a class from the problem domain may also be realized by many different classes in the object design. All classes should represent the same domain concept, but each representation has a different levels of detail. Changing a Model Element on a specific abstraction level may require to change the related Model Elements on other abstraction levels as well. Korbinian Herrmann is currently extending the RUSE model with an abstraction layer concept and evaluates his ideas with Sysiphus . The layers form a hierarchy and each layer is connected by traceability links to any related Model Elements of the RUSE model. Depending on the Model Element type, the mechanism tries to automatically propagate changes from one layer to the next. His concepts should also facilitate the release management of system model elements. Initial results are published in [57].

The research of Anil Kumar Thurimella focuses on issue-based variability modeling for product line engineering. The issue-based variability modeling supports the instantiation and evolution of the variations in product lines as well as

enabling informal collaboration in software product line engineering. He integrates our existing issue model and the capabilities for informal collaboration with traditional variability modeling. Initial results are published in [122].

This dissertation focuses on the integration of different models and distributed project collaboration. New research needs to investigate how the system implementation and the source code can be integrated or benefit from our concepts. Also the rationale behind source code gets usually lost in long running projects. Source code documentation gets outdated very fast and new programmers have to investigate the code frequently. Rationale management integrated into the development IDEs would probably help to overcome these problems. Traceability across classes, attributes and methods are supported by the compiler, but traceability to related system models like use cases or to design issues is still a problem.

APPENDIX A

Bibliography

[1] R.J. Abbott. Program design by informal english description. *Comm. of the ACM*, 26(11):82–94, 1983. 52

[2] Annie I. Anton. Goal-based requirements analysis. In *IEEE International Conference on Requirements Engineering (ICRE '96)*, pages 136–144, Colorado Springs, Colorado, USA, April 1996. Available from: citeseer.nj.nec.com/article/anton96goalbased.html. 12

[3] Annie I. Anton, R.A. Carter, A. Dagnino, J.H. Dempster, and D.F. Siege. Deriving goals from a use case based requirements specification. *Requirements Engineering Journal*, 6:63–73, May 2001. 12

[4] Apache. Apache derby [online]. April 2007. Available from: http://db.apache.org/derby/. 91, 93

[5] ARENA [online]. January 2006. Available from: http://sysiphus.informatik.tu-muenchen.de/arena/. 115

[6] Robert D. Battin, Ron Crocker, Joe Kreidler, and K. Subramanian. Leveraging resources in global software development. *IEEE Software*, pages 70–77, March/April 2001. 11, 78, 121

[7] Kent Beck. *Extreme Programming Explained: Embrace Chang*. Addison-Wesley, 1999. 12

[8] Brian Berenbach. Evaluating the quality of a uml business model. In *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 280, Washington, DC, USA, 2003. IEEE Computer Society. 21

[9] Brian Berenbach. The evaluation of large, complex uml analysis and design model. In *ICSE '04: Proceedings of the 26th International Conference on*

Software Engineering, pages 232–241, Washington, DC, USA, 2004. IEEE Computer Society. 12, 21

[10] Brian Berenbach. Impact of organizational structure on distributed requirements engineering processes: lessons learned. In *GSD '06: Proceedings of the 2006 international workshop on Global software development for the practitioner*, pages 15–19, New York, NY, USA, 2006. ACM Press. 14

[11] Brian Berenbach. Introduction to product line requirements engineering. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, page 215, Washington, DC, USA, 2006. IEEE Computer Society. 14

[12] Brian Berenbach and Gail Borotto. Metrics for model driven requirements development. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 445–451, New York, NY, USA, 2006. ACM Press. 12, 22, 49, 129

[13] Brian Berenbach and Mark Gall. Toward a unified model for requirements engineering. In *Proceedings of the First International Conference on Global Software Engineering*, pages 237–238, 2006. 12, 49

[14] Barry Boehm, Paul Grunbacher, and Robert O. Briggs. Easywinwin: A groupware-supported methodology for requirements negotiation. *icse*, 00:0720, 2001. 119

[15] Borland. Together [online]. April 2007. Available from: http://www.borland.com/us/products/together. 12

[16] Lars Borner and Barbara Paech. Teaching the software engineering process emphasizing testing, rationale and inspection (train). In *European Symposium on Systems Analysis and Design: Practice and Education*, June 2006. 117

[17] Lars Borner, Barbara Paech, and Jürgen Rückert. Vom modellverstehen zum modell-erstellen. In *Modellierung 2006, Workshop Modellierung in Lehre und Weiterbildung*, 2006. 117

[18] Robert O. Briggs and Paul Gruenbacher. Easywinwin: Managing complexity in requirements negotiation with gss. In *35th Hawaii International Conference on System Sciences*, volume 1, page 21b, 2002. 119

[19] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, Englewood Cliffs, NJ, second edition, Sep 2003. Available from: http://wwwbruegge.in.tum.de/OOSE/WebHome. 11, 12, 20, 24, 28, 52, 53, 58, 81, 95, 96, 115, 152, 154

[20] Bernd Bruegge, Allen H. Dutoit, Rafael Kobylinski, and Günter Teubner. Transatlantic project courses in a university environment. In *Asian Pacific Software Engineering Conference*, Dec 2000. 101

[21] Bernd Bruegge, Allen H. Dutoit, and Timo Wolf. Sysiphus: Enabling informal collaboration in global software development. In *Proceedings of the First International Conference on Global Software Engineering*, October 2006. Available from: http://www.icgse.org/. 15, 81

[22] Chair for Applied Software Engineering, Technische Universität München. Cargo & Logistic [online]. 2003. Available from: http://wwwbruegge.informatik.tu-muenchen.de/Logistic/WebHome. 103

[23] Chair for Applied Software Engineering, Technische Universität München. CampusTV [online]. 2004. Available from: http://wwwbruegge.informatik.tu-muenchen.de/SoftwareEngineeringPraktikumWiSe2004. 104

[24] James Chisan and Daniela Damian. Towards a model of awareness support of software development in gsd. In *The 3rd International Workshop on Global Software Development*, pages 28–33, May 2004. 20, 121

[25] Jeff Conklin and K. C. Burgess-Yakemovic. A process-oriented approach to design rationale. *Human-Computer Interaction*, 6(11):357–391, 1991. 61

[26] Reidar Conradi and Bernhard Westfechtel. Towards a uniform version model for software configuration management. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 1–17, London, UK, 1997. Springer-Verlag. 32, 39

[27] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998. 32, 40

[28] Oliver Creighton, Christoph Angerer, Timo Wolf, Allen H. Dutoit, and Bernd Bruegge. Temporary roles: An explicit, user-specified organizational model. In *First Workshop on Pervasive Security, Privacy and Trust (PSPT)*, Aug 2004. Available from: http://www.pspt.org/. 87

[29] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Comm. ACM*, 31(11), November 1988. 13

[30] Daniela Damian, Filippo Lanubile, and Teresa Mallardo. Investigating ibis in a distributed educational environment: the design of a case study. In *Int'l Workshop on Distributed Software Development*, pages 153–158, August 2005. 119, 121

[31] Susan Dart. Spectrum of functionality in configuration management systems. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990. 23

[32] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, New York, NY, USA, 1991. ACM Press. 23

[33] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In *CSCW*, pages 63–71, New York, NY, USA, 2004. 20, 119, 120

[34] Barbara Dellen, Kirstin Kohler, and Frank Maurer. Integrating software process models and design rationales. In *Knowledge-Based Software Engineering Conference*, volume 11, 1996. 122

[35] Jeremy Dick. Design traceability. *IEEE Software*, 22(6):14–16, November/December 2005. 18, 50, 120

[36] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *Conference Proceedings on Computer-supported Cooperative Work*, 1992. 20, 78

[37] Allen H. Dutoit, Raymond McCall, Ivan Mistrik, and Barbara Paech, editors. *Rationale Management in Software Engineering*. Springer Verlag, April 2006. 59, 119

[38] Allen H. Dutoit and Barbara Paech. Rationale-based use case specification. *Requirements Engineering Journal*, 7(1):3–19, 2002. Available from: http://link.springer.de/link/service/journals/00766/tocs/t2007001.htm. 15, 81

[39] Allen H. Dutoit, Timo Wolf, Barbara Paech, Lars Borner, and Jurgen Ruckert. Using rationale for software engineering education. In Timothy C. Lethbridge and Daniel Port, editors, *CSEET '05: Proceedings of the 18th Conference on Software Engineering Education & Training*, pages 129–136, Washington, DC, USA, April 2005. IEEE Computer Society. Available from: http://www.site.uottawa.ca/cseet2005. 101

[40] Christof Ebert and Philip De Neve. Surviving global software development. *IEEE Software*, pages 62–69, March/April 2001. 11, 78, 121

[41] Jacky Estublier, David Leblang, Andre van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, 2005. 23

[42] Eclipse Foundation. Eclipse [online]. 2007. Available from: http://www.eclipse.org. 111

[43] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. 38

[44] Free Software Foundation. Gnu general public license. http://www.gnu.org/copyleft/gpl.html, April 2007. 83, 115

[45] Free Software Foundation (FSF). The revision control system. http://www.gnu.org/software/rcs/rcs.html, October 2006. 39

[46] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. 35, 84, 147

[47] Orlena Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In *International Conference on Requirements Engineering*, pages 94–101, Colorado, April 1994. IEEE. Available from: citeseer.ist.psu.edu/gotel94analysis.html. 18, 50, 59, 120

[48] Orlena Gotel and Anthony Finkelstein. Contribution structures. In *International Symposium on Requirments Engineering*, pages 100–107. IEEE, March 1995. Available from: citeseer.ist.psu.edu/article/gotel95contribution.html. 18, 50, 108

[49] Orlena Gotel and Anthony Finkelstein. Extended requirements traceability: Results of an industrial case study. In *3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pages 169–179. IEEE Computer Society, 1997. Available from: citeseer.ist.psu.edu/gotel97extended.html. 18

[50] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall Press, 1992. 23

[51] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The geography of coordination: Dealing with distance in R&D work. *ACM*, 1999. 13

[52] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, New York, NY, USA, 2004. ACM Press. 20

[53] James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, June 2003. 13, 20

[54] James D. Herbsleb, Audris Mockus, Thomas A. Finholt, and Rebecca E. Grinter. An empirical study of global software development: Distance and speed. *icse*, 00:0081, 2001. 20

[55] Andrea Herrmann and Barbara Paech. Software quality by misuse analysis. Technical Report SWEHD-TR-2005-01, University of Heidelberg, http://www-swe.informatik.uni-heidelberg.de/research/publications/reports.htm, 2005. 113

[56] Andrea Herrmann and Barbara Paech. Moqare = "misuse-oriented quality requirements engineering" – über den nutzen von bedrohungsszenarien beim re von qualitätsanforderungen. *Softwaretechnik-Trends*, 26(1), February 2006. 113

[57] Korbinian Herrmann and Bernd Bruegge. Visualization of release planning. In *Proceedings of the International Workshop on Requirements Engineering Visualization (REV 2006)*, September 2006. 129

[58] Florian Huber. Design and implementation of an awareness system, integrating sysiphus and eclipse. Diplomarbeit, Technische Universität München, 2006. 111

[59] IBM. Rational ClearCase [online]. Mar 2007. Available from: http://www-306.ibm.com/software/awdtools/clearcase. 12, 22, 39

[60] IBM. Rational ClearQuest [online]. April 2007. Available from: http://www-306.ibm.com/software/awdtools/clearquest. 12

[61] IBM. Rational RequistePro [online]. Mar 2007. Available from: http://www-306.ibm.com/software/awdtools/reqpro. 12, 125

[62] IBM. Rational Rose [online]. April 2007. Available from: http://www-306.ibm.com/software/awdtools/developer/rose. 12, 106

[63] IBM. Rational Software Architect [online]. Mar 2007. Available from: http://www-306.ibm.com/software/awdtools/architect/swarchitect/. 12, 125

[64] IBM. Rational Software Modeler [online]. Mar 2007. Available from: http://www-306.ibm.com/software/awdtools/modeler/swmodeler. 20, 89

[65] IEEE. IEEE guide to software configuration management. *ANSI/IEEE Std 1042-1987*, 1987. 22

[66] IEEE. *IEEE Standard for Software Configuration Management Plans*. June 1998. 22

[67] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999. 23

[68] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1994. 12, 51

[69] Matthias Jarke. Requirements tracing. *Comm. ACM*, 41(12):32–36, December 1998. 18, 50

[70] Kyo C Kang, Sholom G. Cohen, James A.Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report SEI-90-TR-21, CMU, 1990. Available from: http://www.sei.cmu.edu/domain-engineering/FODA.html. 50, 149

[71] Laurent Karsenty. An empirical evaluation of design rationale documents. In *CHI '96*, pages 150–156, 1996. 59

[72] Rafael Kobylinski, Oliver Creighton, Allen H. Dutoit, and Bernd Bruegge. Building awareness in distributed software enginering: Using issues as context. In *International Workshop on Distributed Software Development, International Conference on Software Engineering*, May 2002. 20

[73] R. E. Kraut and L. A. Streeter. Coordination in software development. *Comm. ACM*, 38(3), Mar 1995. 13

[74] Filippo Lanubile, Teresa Mallardo, and Fabio Calefato. Tool support for geographically dispersed inspection teams. *Software Process: Improvement and Practice*, 8(4):217–231, October/December 2003. 119, 121

[75] Kim Letkeman. Comparing and merging uml models in ibm rational software architect: Part 1 - comparing models with local history. Technical report, Modeling Compare Support, IBM Rational, 2005. 33, 125

[76] Kim Letkeman. Comparing and merging uml models in ibm rational software architect: Part 2 - merging models using "compare with each other". Technical report, Modeling Compare Support, IBM Rational, 2005. 33, 125

[77] Kim Letkeman. Comparing and merging uml models in ibm rational software architect: Part 3 - a deeper understanding of model merging. Technical report, Modeling Compare Support, IBM Rational, 2005. 32, 33, 125

[78] Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press. 33, 123

[79] Andrea De Lucia, Fausto Fasano, Rita Francese, and Rocco Oliveto. Traceability management in adams. In *Int'l Workshop on Distributed Software Development*, pages 125–139, Aug. 2005. 119, 120

[80] Allan MacLean, Richard M. Young, Victoria M.E. Bellotti, and Thomas P. Moran. Questions, options, and criteria: Elements of design space analysis. *HCI*, 6(3-4):201–250, 1991. 15, 61, 62, 119, 122, 128

[81] Mariss Jansons [online]. April 2007. Available from: http://www.br-online.de/kultur-szene/klassik/pages/so/so_chefdirigent.html. 108

[82] Akhil Mehra, John Grundy, and John Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2005. ACM Press. 32, 124

[83] Mercury. Testdirector [online]. April 2007. Available from: http://www.mercury.com/us/products/quality-center/testdirector. 12

[84] Microsoft. Microsoft office word [online]. April 2007. Available from: http://office.microsoft.com/word. 12, 20

[85] Audris Mockus and James D. Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *International Conference on Software Engineering*, pages 503–512, May 2002. 20, 122

[86] Mozilla. Bugzilla [online]. 2007. Available from: http://www.bugzilla.org/. 111

[87] MySQL. Mysql community server [online]. April 2007. Available from: http://www.mysql.com/. 92, 94

[88] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. Flexible fine-grained version control for software documents. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 212–219, Washington, DC, USA, 2004. IEEE Computer Society. 124

[89] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. Configuration management for designs of software systems. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 236–243, Washington, DC, USA, 2005. IEEE Computer Society. 124

[90] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224, New York, NY, USA, 2005. ACM Press. 23, 33, 40, 124

[91] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000. 19, 21

[92] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *The Journal of Systems and Software*, 58(2):171–180, 2001. Available from: citeseer.ist.psu.edu/nuseibeh01making.html. 19, 21

[93] Object Management Group, Inc. *Unified Modeling Language Specification Version 2.0*, May 2004. 12, 15, 25, 51

[94] Object Management Group, Inc. *Object Constraint Language Specification, version 2.0*, 2006. Available from: http://www.omg.org/technology/documents/formal/ocl.htm. 25

[95] Takafumi ODA and Motoshi SAEKI. Meta-modeling based version control system for software diagrams. *IEICE Trans Inf Syst*, E89-D(4):1390–1402, 2006. Available from: http://ietisy.oxfordjournals.org/cgi/content/abstract/E89-D/4/1390. 33, 124

[96] D. Ohst. A fine-grained version and confguration model in analysis and design. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 521, Washington, DC, USA, 2002. IEEE Computer Society. 32, 123

[97] Dirk Ohst, Michael Welle, and Udo Kelter. Difference tools for analysis and design documents. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 13, Washington, DC, USA, 2003. IEEE Computer Society. 123

[98] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press. 123

[99] Hamilton Oliveira, Leonardo Murta, and Claudia Werner. Odyssey-vcs: a flexible version control system for uml model elements. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 1–16, New York, NY, USA, 2005. ACM Press. 33, 124

[100] Barbara Paech, Lars Borner, Jürgen Rückert, Allen H. Dutoit, and Timo Wolf. Vom Kode zu den Anforderungen und zurück: Software Engineering in 6 Semesterwochenstunden. In *Software Engineering im Unterricht der Hochschulen. Aachen, 2005 (SEUH 2005)*, 2005. 117

[101] Dewayne E. Perry, Adam A. Porter, Michael W. Wade, Lawrence G. Votta, and James Perpich. Reducing inspection interval in large-scale software development. *IEEE Trans. Soft. Eng.*, 28(7):695–705, 2002. 121

[102] Dewayne E. Perry, Nancy Staudenmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994. 13

[103] Balasubramaniam Ramesh and Vasant Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Trans. Softw. Eng.*, 18(6):498–510, 1992. 122

[104] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, 2001. 18

[105] Jungkyu Rho and Chisu Wu. An efficient version model of software diagrams. In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 236, Washington, DC, USA, 1998. IEEE Computer Society. 33, 123

[106] Matthias Riebisch. Supporting evolutionary development by feature models and traceability links. *ecbs*, 00:370, 2004. 51

[107] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA, June 2002. 50

[108] Rohde & Schwarz [online]. April 2007. Available from: http://www.rohde-schwarz.de/. 104

[109] Jürgen Rückert and Barbara Paech. Software engineering moderner anwendungen. In *Software Engineering im Unterricht der Hochschulen (SEUH)*, pages 59–72, 2007. 118

[110] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, New York, NY, 1991. 84

[111] Anita Sarma and Andre van der Hoek. Towards awareness in the large. In *Proceedings of the First International Conference on Global Software Engineering*, October 2006. 20

[112] Jennifer Schiller. Design and implementation of the agile project management method scrum in sysiphus. Master's thesis, Technische Universität München, 2007. 113

[113] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2001. 112

[114] SEI. Capability maturity model integration [online]. 2006. Available from: http://www.sei.cmu.edu/cmmi. 22, 31, 120

[115] Artisan Software. Artisan studio home, November 2006. Available from: http://www.artisansw.com. 126

[116] Software Engineering Group at the University of Heidelberg [online]. 2007. Available from: http://www-swe.informatik.uni-heidelberg.de/. 113, 117

[117] StandishGroup. Extreme chaos [online]. 1999. Available from: http://standishgroup.com/sample_research. 120

[118] M. A. Storey, Davor Cubranic, and D. M. German. On the use of visualization to support awareness of human activities in software development:a survey and a framework. In *Proceedings of the 2nd ACM Symposium on Software Visualization*, 2005. To be presented. 20

[119] Symphonieorchester des Bayerischen Rundfunks [online]. April 2007. Available from: http://www.br-online.de/kultur-szene/klassik_e/pages/so/. 108

[120] TEAM [online]. 2006. Available from: http://www.team-project.eu/. 114

[121] Telelogic. Doors [online]. Mar 2007. Available from: http://www.telelogic.com/products/doors. 12, 18, 20, 23, 107, 125

[122] Anil Kumar Thurimella. Rationale-based variability management in product line requirements engineering. In *International Conference on Software Engineering*. IASTED, 2007. 130

[123] Tigris.org. Subversion version control system [online]. Mar 2007. Available from: http://subversion.tigris.org. 12, 22, 39

[124] Valentino Vranic. Reconciling feature modeling: A feature modeling metamodel. In *Net.ObjectDays*, pages 122–137, 2004. 51

[125] WACKER Chemie AG [online]. April 2007. Available from: http://www.wacker.com. 103

[126] Timo Wolf. Design and implementation of a rationale-based analysis tool. Diplomarbeit, Technische Universität München, 2002. 81

[127] Timo Wolf and Bernd Bruegge. Virtual symphony orchestra [online]. January 2005. Available from: http://wwwbruegge.in.tum.de/VSO/. 108

[128] Timo Wolf and Allen H. Dutoit. A rationale-based analysis tool. In Walter Dosch and Narayan Debnath, editors, *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, pages 209–214. ISCA, July 2004. Available from: http://www.isp.uni-luebeck.de/iasse04/index.htm. 15, 53, 81

[129] Timo Wolf and Allen H. Dutoit. Sysiphus: Combining system modeling with collaboration and rationale. *Softwaretechnik-Trends*, 24(4), November 2004. Available from: http://pi.informatik.uni-siegen.de/stt/24_4/. 60

[130] Timo Wolf and Allen H. Dutoit. Supporting traceability in distributed software development projects. In *Proceedings of the International Workshop on Distributed Software Development*, pages 111–124, August 2005. 77

[131] Timo Wolf and Allen H. Dutoit. Sysiphus at http://sysiphus.in.tum.de [online]. May 2005 [cited 27.05.2005]. Available from: http://sysiphus.in.tum.de. 15, 81, 154

[132] Yieeha Ltd. Co KG [online]. 2007. Available from: http://www.yieeha.de. 113

APPENDIX B

List of Figures

APPENDIX C

Listings

APPENDIX D

Glossary

A

Abstract Operation The Abstract Operation class is the abstract super class of the Meta Model Operation and Model Operation classes. The Abstract Operation class and its subclasses build a composite pattern [46], in which the Abstract Operation describes an abstract change., p. 35.

Actor An Actor represents the role of an external person of system that interacts with the system under development. The interaction is described in Use Cases., p. 51.

Admin Subsystem The Admin Subsystem provides a graphical user interface for administrative tasks of the Repository Subsystem., p. 89.

Annotation In the RUSE model, users collaborate by linking collaboration artifacts, called Annotations, to Model Elements.

API (application programming interface) A set of definitions of the ways in which one piece of computer software communicates with another. It is a method of achieving abstraction, usually (but not necessarily) between lower-level and higher-level software. Definition adapted from [http://en.wikipedia.org/wiki/API]., p. 29.

Assessment Assessments represent the evaluation of a single Proposal against a Criterion, p. 61.

Assignable The Assignable class represent any item in a project that needs a responsible Organizational Unit., p. 49.

B

Branch The Branch class represents a branch of concurrent development in the version space and is composed of all versions that are available in the branch., p. 24.

C

Cause The Cause class describes the circumstances that leads to its associated Hazard, p. 55.

Change Package The Change Package represents and encapsulates the changes on the Project Data, the Model Elements, and the Model Links between the two Versions that are connected by the associated Revision Link. A Change Package represents the differences or deltas between the versions. It is also used to create a new successor version for an existing version., p. 25.

Client Application Layer The Client Application Layer contains the subsystems of the Sysiphus client applications. The subsystems include the RAT Subsystem, the REQuest Subsystem, the Notification Subsystem, the SubClipse Subsystem, and the Admin Subsystem., p. 4.

Comment Comments are an informal and unstructured way for project participants to communicate, similar to posts in a newsgroup. The Comment class extends the Annotation class., p. 60.

Communication Subsystem The Communication Subsystem is responsible for transporting remote service calls, their corresponding data and results between the Workspace Subsystem and the Repository Subsystem.

Composite Section The Composite Section is part of a Document and can have any subsections in a Section hierarchy.

Criterion The Criterion class represents desirable qualities that selected Proposals should satisfy.

D

Derby (Apache Derby) Apache Derby is part of the Apache DB subproject, and is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0. [http://db.apache.org/derby], p. 91.

Diagram The Diagram class of the RUSE model is used to represent diagrams containing Model Elements. Vertex instances are used to defined which Model Elements are part of aDiagram.

Document The Document class represent any documents in the RUSE model and consists of Sections.

E

Element Store Layer The subsystems in the Element Store Layer provide the Rationale-based unified software engineering model Meta-Model and underlying services for the Rationale-based unified software engineering model Model, including access control, persistent storage, and a central server repository that provides Project access to distributed client applications., p. 4.

F

Feature A Feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [70]., p. 50.

Functional Requirement (FR) A Functional Requirement describes a required function of the system under development. The granularity must be sufficiently fine that the requirement can be tested against a component realizing the requirement., p. 54.

H

Hazard The Hazard class describes a potential harm or injury of a user, which might occur when using the system., p. 55.

History The History class represents the history of a project. It provides operations for creating revisions, branches, tags and for accessing specific versions, differences and history information., p. 24.

History Link The History Link class defines the successor Versions for a given Version and represents the edges in the version graph. The History Link has the two subclasses Revision Link and Variant Link that are defined in 2.2.2. Revision Links represent changes between two Versions within the same branch, while the Variant Link connects two Versions from different branches., p. 24.

HTML (HyperText Markup Language) A coding system that provides a standard for integrating graphics, multimedia, and references to distant texts in WWW documents., p. 89.

HTTP (HyperText Transfer Protocol) The communications protocol of the WWW., p. 90.

I

Issue The Issue represents critical problem that has no clear solution. Different Proposals are evaluated before formulating a Resolution, p. 49.

L

Leaf Section The Leaf Section class represent the leafs of a Section hierarchy in a Document and has a filter to include any Model Elements.

M

Meta Model Operation Meta Model Operations describe changes on the sematic level of the meta-model. They affect exactly one Model Element and describe a single change. A Meta Model Operation support reversibility as described in section 2.2.2., p. 34.

Mitigation The Mitigation class associates Requirements with Hazards or Causes. The Requirements were created to mitigate the associated Hazards or Causes, p. 56.

Model Layer The Model Layer provides the domain knowledge of software engineering artifacts and models. It consists of the Model Subsystem and the Model Service Subsystem., p. 4.

Model Service Subsystem The Model Service Subsystem provides common services on the Rationale-based unified software engineering model model, provided by the Model Subsystem, that are shared among several client applications of Sysiphus., p. 88.

Model Subsystem The Model Subsystem provides the RUSE model as described in chapter 3., p. 88.

Model Element The class Model Element is the most abstract and generic modeling class that represents any concept of the software engineering domain. A model element can have many child model elements and can be linked by Model Link classes to many other model elements. The Model Element class provides generic methods to store and retrieve arbitrary data., p. 25.

Model Link The Model Link class is an association class that links two related Model Element instances. The Model Link class extends the Model El-

ement class and inherits to all its properties to represent a software engineering concept., p. 25.

Model Operation The Model Operation describes changes on the semantic level of the model layer., p. 35.

N

Nonfunctional Requirement (NFR) A Nonfunctional Requirement describes a required function of the system under development. The granularity must be sufficiently fine that the requirement can be tested against a component realizing the requirement., p. 54.

Notification Subsystem The Notification Subsystem sends notification emails about user-relevant changes of a Project to the end-users., p. 89.

O

Object Constraint Language (OCL) A declarative language for describing rules that apply to UML models developed at IBM and now part of the UML standard. Definition adapted from [http://en.wikipedia.org/wiki/OCL]., p. 25.

Organizational Unit The RUSE model contains an organizational model consisting of Organizational Units that are either Participants or Teams., p. 47.

P

Participant The Participant represent a person that is part of the organizational model of a project., p. 47.

Project A class Project represents a software development project, containing all project related entities. It provides access to the project data, the history containing different versions and their changes, as well as to the users that access the project., p. 24.

Project Data Subsystem The Project Data Subsystem realizes the project data model described in section 2.2.1 and provides the services necessary to access and manipulate the meta-model.

Project Data The class Project Data represents the complete data of a project in a specific version. It is composed of Model Elements and Model Links, which relates two elements, thus building a graph structure. The Project Data class provides operations for accessing, searching and filtering model elements., p. 25.

Proposal Proposals are possible solutions for associated Issues., p. 61.

R

Requirements Analysis Document (RAD) A document describing the analysis model of the problem domain., p. 58.

RAT Subsystem The RAT Subsystem provides the end-user desktop application RAT, a graphical user interface for accessing and manipulating the RUSE Model., p. 89.

RDF (Resource Description Framework) A language designed to support the Semantic Web, in much the same way that HTML is the language that helped initiate the original Web. The RDF supports resource description, or metadata (data about data), for the Web. It provides common structures that can be used for interoperable XML data exchange., p. 152.

Repository Subsystem The Repository Subsystem provides a Facade to the shared server repository, containing the Projects and its meta-models. It offers all service operations to retrieve and change the meta-model on a server node.

REQuest Subsystem The REQuest Subsystem provides the web-based client application REQuest, which supports a HTML-based hypertext view of the RUSE model., p. 89.

Requirement The Requirement class is used to describe functionality or qualities of the system under development. The granularity must be sufficiently fine that the Requirement can be tested against any components realizing the requirement. The Requirement class is abstract and is the super class of the Functional Requirement and Nonfunctional Requirement classes., p. 54.

Resolution The Resolution class represents the solution of Issues and is based on one or more Proposals., p. 61.

Revision Link The Revision Link class is a subclass of the History Link and represent changes between two Versions within the same Branch. The changes between two versions are encapsulated in the associated Change Package., p. 25.

Role The Role association class relates many Users with many Projects. The Role defines the user's role in a project. Typical roles are *Project Manager*, *Requirements Engineer*, *Analysist*, or *Architect* [19]., p. 24.

RSS (RDF Site Summary or Rich Site Summary) A mechanism to publish and subscribe to news feeds, like e.g. websites and so-called weblogs. Not to be confused with the competing plain-XML based standard *Really Simple Syndication*, p. 156.

Rationale-based unified software engineering model (RUSE) A model unifying communication, software development models and project management with rationale for distributed software development projects, p. 14.

S

software configuration management (SCM) The purpose of Software Configuration Management is to establish and maintain the integrity of the products of the software project throughout the project's software lifecycle. Software Configuration Management involves identifying the configuration of the software (i.e., selected software works products and their descriptions) at given points in time, systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the software lifecycle. The work products placed under software configuration management include the software products that are delivered to the customer (e.g., the software requirements document and the code) and the items that are identified with or required to create these software products (e.g., the compiler). A software baseline library is established containing the software baselines as they are developed. Changes to the baselines and the release of software products built from the baseline library are systematically controlled via the change control and configuration auditing functions of software configuration management., p. 22.

System Design Document (SDD) The System Design Document is a document describing the system design model., p. 58.

Section The Section class is the superclass of the classes Composite Section and Leaf Section and consists of a title, text, and many Diagrams.

Stakeholder The Stakeholder class represents a person or organization who has a certain interest in the Project., p. 48.

Stakeholder Request A Stakeholder Request class is used to describe the needs and requests of Stakeholders in a Project. They can be accepted or rejected., p. 50.

Storage Management Subsystem The Storage Management Subsystem provides persistency for project data model and the version object model.

SubClipse Subsystem The SubClipse Subsystem provides a framework for developing Eclipse plugins, which access the Repository Subsystem., p. 89.

SVG (Scalable Vector Graphics) A XML based open W3C standard describing two-dimensional vector graphics, both static and animated, p. 156.

Sysiphus Sysiphus [131] is a distributed tool suite that implements the RUSE model. Sysiphus provides a central repository for the RUSE model and provides online access for synchronous collaboration of project participants from distributed sites, as well as offline workspace support for asynchronous interaction., p. 15.

system design model A high-level description of the system, including design goals, subsystem decomposition, hardware/software platform, persistent storage strategy, global control flow, access control policy, and boundary condition strategies. The system design model represents the strategic decisions made by the architecture team that allow subsystem teams to work concurrently and cooperate effectively [19]., p. 153.

T

Tag The Tag class identifies an user defined Version of the version graph by a user defined name., p. 42.

Team A Team consists of Organizational Units that are belonging together., p. 47.

U

UML (Unified Modeling Language) A graphical and formal modeling language, which supports twelve diagram types in three categories: structural, behavioral, and model management. UML helps visualization, specification, construction, and documentation of artifacts of a software-intensive system. Several different predecessors of UML, which were established for object-oriented design, have been unified to simplify graphical modeling and to facilitate exchange., p. 12.

URL (Uniform Resource Locator) , p. 70.

Use Case A Use Case describes the interaction of Actors with the system under development in a textual flow of events.

User The class User represent persons or external entities, accessing a Project. A User may access many Projects, defined by the Role association., p. 24.

User Management Subsystem The User Management Subsystem provides services for authentication, authorization and the management of Users.

V

Variant Link The Variant Link class is a subclass of the History Link and represent changes between two Versions across two different Branches., p. 25.

Version Versions represent the nodes in the version graph. The state of the project at a specific version can either be represented explicitly by a Project Data or implicitly by its position in the version graph and the appropriate deltas, represented by the Change Packages., p. 24.

Version Model Subsystem The Version Model Subsystem provides the version model as described in chapter 2. It provides services for creating, managing and retrieving history information for a Project., p. 86.

Vertex The Vertex class extends the Model Link class and defines which Model Elements are part of a Diagram.

W

W3C (World Wide Web Consortium) The consortium producing and governing the standards of the WWW. It is headed by Tim Berners-Lee, the original creator of URL, HTTP and HTML, the principal technologies that form the basis of the Web, p. 154.

Work Item The Work Item describes the work to done in development project., p. 49.

Workspace Subsystem The Workspace Subsystem provides the interface for accessing and manipulating the projects of the Repository Subsystem to higher layer subsystems, especially to the end-user client applications.

WWW (World Wide Web) A system that provides relatively uniform standards for widely scattered information services on the Internet, including an addressing scheme that permits HyperText references to other sites., p. 90.

X

XML (eXtensible Markup Language) A W3C-recommended general-purpose markup language for creating special-purpose markup languages (it is a metaformat). It is a simplified subset of SGML, capable of describing many different kinds of data. Its primary purpose is to facilitate the sharing of structured text and information across the Internet. Languages based on XML (for example, RDF, RSS, and SVG) are themselves described in a formal way, allowing programs to modify and validate documents in these languages without prior knowledge of their form. Definition adapted from [http://en.wikipedia.org/wiki/XML]., p. 32.