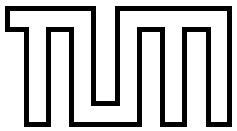
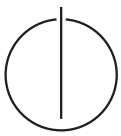


Technische
Universität München
Fakultät für Informatik



Forschungs- und Lehreinheit I
Angewandte Softwaretechnik

The QUARC Metamodel: A Communication-Based Generic Project Model

Michael Nagel

Vollständiger Abdruck der von der promotionsführenden Einrichtung 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. Helmut Krcmar
Prüfer der Dissertation:	1.	Univ.-Prof. Bernd Bruegge, Ph.D.
	2.	Univ.-Prof. Dr. Kurt Schneider, Leibniz Universität Hannover

Die Dissertation wurde am 27.09.2012 bei der Technischen Universität München eingereicht und durch die promotionsführende Einrichtung Fakultät für Informatik am 19.11.2012 angenommen.

Acknowledgements

I am indebted to *Bernd Bruegge*. You have the rare ability to make people around you excel and push their limits. Without your vision and challenges, this dissertation would not have been possible. Working with you has been a great honor.

I would also like to thank my second reviewer, *Kurt Schneider*. Your comments sparked critical changes and extensions.

It appears that trying to understand even a single paragraph or figure in an OMG specification drives a lesser man one step closer to insanity, or to a simple life of hard work and repentance in a monastery, or possibly both. *Florian Schneider* admirably braved the task of reading and understanding not only one OMG specification, but several, without either of these fates befalling him. I am grateful that you freely share your acquired wisdom in more easily digestible form, and that you were always up for critical discussions about new ideas and concepts.

Thank you to the Managing Partners at Linova – *Andreas Löhr*, *Horst Mauersberg*, and *Tobias Weishäupl*. You created a work environment that allowed me to pursue this dissertation in parallel to my everyday work activities.

My gratitude goes to those people who have contributed ideas, rebuttals, insights, support, and generally their valuable time: You know who you are.

I dedicate this dissertation to my significant other, *Lena*. Your constant support and motivation was essential to complete this work. You made me a better person along the way.

Abstract

Project managers use several different tools to handle task management, issue tracking, requirements management, rationale management, risk management, software lifecycle, and decision support systems. These tools have different representations and data models, which makes connecting their artifacts difficult.

We present the QUARC metamodel (QUestions, Activities, Rationale, Communication) which offers a common foundation that allows a unified treatment of the different aspects of these tools. In particular, we show how the metamodel can serve as the basis for a tool that combines software life cycle, task management, rationale management, and risk management.

The QUARC metamodel defines only one entity, the quarc. A quarc represents a task that needs to be done, or a decision that needs to be made. Dependencies between quarcs are represented by directed typed relations. Communication media (images, sound, video, drawings, etc.) can be linked directly to quarcs. The QUARC metamodel treats human interaction elements, i.e. communication, as first-class citizens in project models. Quarc repositories serve as containers for quarcs, communication, and relation definitions. The Quarc Query Language (QQL) allows users to perform operations and specify validators on a quarc repository, and define new attributes and relation types for quarcs.

We show how commonly used models like IBIS, QOC, PERT/CPM, Waterfall, Unified Process, and Scrum can be represented in Quarc repositories. Furthermore, we show how aspects of these models can be combined into a hybrid model due to the common foundation of their QUARC representations.

Projektmanager nutzen verschiedene Werkzeuge, um verschiedene Aspekte eines Projekts wie Task-Management, Issue Tracking, Requirements, Rationale Management, Risikomanagement, Softwarelebenszyklus oder Decision Support abzudecken. Diese Werkzeuge haben unterschiedliche Repräsentationen und Datenmodelle. Die Verbindung der Artefakte aus verschiedenen Werkzeugen ist daher schwierig.

Wir stellen das QUARC Metamodell (QUestions, Activities, Rationale, Communication) vor, das eine einheitliche Grundlage bietet, die es erlaubt, die verschiedenen Aspekte dieser Werkzeuge vereinheitlicht zu behandeln. Insbesondere zeigen wir, wie das Metamodell als Grundlage für ein Werkzeug dienen kann, das Task-Management, Rationale-Management, Risiko-Management und Softwarelebenszyklus kombiniert.

Das Modell definiert nur eine Entität, das Project Quarc. Ein Quarc repräsentiert eine Aufgabe, die erledigt werden muss, oder eine Entscheidung, die getroffen werden muss. Abhängigkeiten zwischen Quarcs werden durch typisierte gerichtete Beziehungen modelliert. Beliebige Medien (Bilder, Ton, Video, Zeichnungen etc) können direkt an Quarcs gebunden werden. Das Quarc-Metamodell behandelt menschliche Interaktion in Form von an Quarcs gebundene Mediensegmente als "first-class citizens", also als Basiselemente, im Projektmodell. Quarc Repositories dienen als Container für Quarcs, Kommunikation, und Beziehungsdefinitionen. Die Quarc Query Language (QQL) ermöglicht die Ausführung von Operationen und die Spezifikation von Validatoren auf dem Quarc Repository sowie die Definition von neuen Attributen und Beziehungstypen für Quarcs.

Wir zeigen, wie häufig eingesetzte Modelle wie IBIS, QOC, PERT/CPM, Waterfall, Unified Process und Scrum als Quarc Repositories repräsentiert werden können. Darüber hinaus zeigen wir, wie Aspekte dieser Modelle durch die gemeinsame Basis ihrer QUARC-Repräsentationen zu hybriden Modellen kombiniert werden können.

Conventions

A sans serif typeface is used to highlight class names, literals and code passages. Multiline code passages are an exception. They are set in a `constant width typeface` to preserve the original indentation.

Inline citations are accentuated by “double quotes.” Extensive citations are additionally set as blocks with a right and a left margin. Changes to direct citations are marked within {curly brackets.}

Contents

CHAPTER 1

Introduction

"Organizations [...] are constrained to produce designs which are copies of the communication structures of these organizations"
- Mel Conway [Con68]

There are many ways to manage software projects. Some run without much control or structure at all, for example self-organizing, community-driven or experimental projects, others are highly structured and follow specific processes and model models rigidly. In general, every project includes activities to define a software lifecycle model, recognize risks and define contingency plans, deal with issues and rationale, provide mechanisms for communication, and manage tasks. The current practice is that project managers use several different tools for these tasks. These tools have different representations and data models, which makes connecting artifacts from different tools difficult.

Software life cycle models define a model for the software development process in a project. Not all these models are well suited to accommodate change in a project. The waterfall model [Roy70], for example, requires that the entire project is split into development activities, which are executed sequentially. This model cannot deal with changes that occur after an activity that they influence has already concluded. For example, if a change that affects requirements is inflicted upon a waterfall project in the implementation phase, then there's no provision in the process how to handle it at that stage. It would have to wait for a maintenance or followup project.

The other end of the spectrum is occupied by agile methods. Agile methods employ iterations and incremental development to cope with change [BBvB+01]. If the change occurs in Scrum [Sch04] project at any time, it is always possible to schedule another sprint to deal with it.

Task management consists of identifying tasks in a work breakdown structure, assigning them to project participants, scheduling their execution, and tracking their progress. The spectrum ranges from formal tools allowing detailed planning and scheduling in advance (PERT

[MRCF59], CPM [KJW59]) to informal methods such as Todo lists (Getting Things Done [All02]) and backlogs (Scrum [Sch04]).

"Those who cannot remember the past are condemned to repeat it"
- George Santayana

Risk management consists of risk assessment (identification, analysis, evaluation) and risk treatment (countermeasures, monitoring, mitigation). Identification turns a risk from an unexpected into an unplanned event. A risk manager mitigates some classes of risks in advance by setting up plans that come into effect when a risk materializes. There are several established risk management processes, such as the approach initially proposed by Barry Boehm [Boe89] [Boe91] or the recent ISO 31000 standard [Int09].

When a decision needs to be reevaluated due to change, e.g. technology change or requirements change, rationale can help to understand the reasons for the earlier decision and to avoid mistakes. Issue management organizes design questions in a project to establish a structure that shows dependencies. Rationale management is often a part of issue management. Iterations and incremental work often requires project participants to revisit and rework the results of earlier decisions and finished tasks. If the rationale for these decisions and discussions about tasks is available, it may contain valuable information for such a revisit that speeds up the process and even helps prevent wrong decisions or dead-end paths. As a result, there are many different approaches and tools to capture and manage rationale ([KR70], [MYBM91], [BR89], [DP02]).

"In the beginning was the Word"
- John 1:1

Communication in a project occurs on many different occasions, in different media, and in different degrees of formality. At one end of the spectrum are documents that follow a rigid model and timeline [Sch00]. This is appropriate for large complex systems, such as safety-critical systems in the aerospace domain [MFM+95]. Creative processes, at the other end, require a different kind of communication. For example, a conversation between two project participants meeting at a water cooler is informal [KSM84] [KFRC90].

To use informal communication in a rationale or task management tool, it needs to be reorganized in a rationale or project management task model. This is currently only possible with high effort. Automated approaches, such as word spotting in recorded conversations [Sch11], require a rhetorical model that does not work well in creative processes that include brainstorming.

To cover the whole spectrum of these activities in a flexible, but unified way, we present the QUARC metamodel *(QUestions, Activities, Rationale, Communication)*. It offers a common foundation for these activities which were previously considered as separate. We show how the metamodel can serve as the basis for task management, rationale management, risk management, and decision support systems. A QUARC model also allows the inclusion of informal communication such as hallway conversations, instant message logs and throwaway sketches. QUARC integrates the media into the project model and makes it accessible to all project participants. Different tools can use the QUARC metamodel to work on a common data model.

An important goal of the QUARC meta model is to support processes and project whose rules and structures may change at runtime. The modification of a life cycle model is known as *tailoring*[Ins06]. This tailoring is usually done at the beginning of a lifecycle definition, be it for a single project or all the projects of an organization. Modification at design time, i.e. before a project or process is executed or instantiated, is called *static tailoring*. A lifecycle model represented by a quarc repository can be modified at runtime. For example, a running software project can be migrated to a different life cycle model, or a running business process can be adapted to changes while it is executed. Modification at runtime is called *dynamic tailoring*.

In the QUARC approach, the metamodel and associated system models can evolve over time from a communication-artifact-based, incomplete, inconsistent, and possibly even incorrect, model to a complete, consistent, correct specification model. In the beginning of a project, the system model is just emerging. Ideas and concepts can be captured in their original form as voice recordings, sketches etc. in a QUARC model. In later phases, the QUARC model is transformed into system model elements. The original ideas and concepts become an annotation to the system model elements as design rationale. The result is a specification model. These specification models are then transformed further into executable code.

We present the Quarc Query Language (QQL) to facilitate interaction with a QUARC model. QQL defines operations to create, read, update and delete (CRUD) elements of the QUARC model. Using QQL, one can set up structural integrity constraints for quarc repositories to enforce behavior or structure patterns. QQL also allows specialized views on a repository, such as partitioning the elements in a QUARC model.

This dissertation is organized as follows:

Chapter 2 describes the QUARC metamodel and its basic entity, the *Quarc*, as well as quarc attributes and quarc relations. We show how the QUARC metamodel handles different kinds of communication and introduce the *Quarc Repository* as a container for quarcs.

Chapter 3 introduces the Quarc Query Language (QQL). Operations on a quarc repository are performed by QQL statements. We define QQL's syntax and describe its system functions for list management, quarc management and attribute/relation type definitions. We also show how structural integrity constraints on quarc repositories can be written as QQL *validators*.

Chapter 4 illustrates the expressiveness of the QUARC model, using two well-known areas of concern for developers and project managers: software development (software lifecycle) and project management (issue management, risk management and task management). We show how widely used models such as IBIS, QOC, PERT/CPM and lifecycle models such as Waterfall, Unified Process, and Scrum can be represented as QUARC models. Furthermore, we present *hybrid models* that combine aspects from these models.

Chapter 5 concludes this dissertation and provides some ideas for extending the QUARC metamodel.

CHAPTER 2

The QUARC Metamodel

In this chapter, we describe the *QUARC* metamodel (*QUestions, Activities, Rationale, Communication*) that allows the instantiation of models for project communication and management for projects with support for frequent change and the incorporation of new technologies during a project. The QUARC metamodel defines a basic entity called *Quarc* that represents a problem that needs to be resolved during the project. These problems range from the very broad, such as "We need to do requirements analysis", to the very narrow such as "Which color should our Cancel buttons have".

2.1 Scope

The goal of the metamodel is to support the instantiation of models for *emergent projects*. We define emergent projects as projects that require a significant amount of design work. The structure of an emergent project can change in unpredictable ways.

For example, any project that builds a new system from scratch requires significant design efforts. Iteration and maintenance projects can also involve design work if there is significant change in the system, such as the incorporation of new technology, or the addition of features that involve several subsystems, or require a modification of the user interface control flow. Iterative process models such as the Unified Process [JBR99] and Scrum [SB01] include design phases in each iteration, i.e. in each workflow and sprint, respectively. Therefore, they are better suited to address frequent change than linear models such as the Waterfall Model [Roy70] and the V-Model [JT79].

The QUARC metamodel is an *entity-based* model with just one core entity, the quarc. Processes are usually not modeled on entities, but on activities, i.e. a process is defined by a collection of activities and transitions between the activities. In the QUARC concept, there are no activities that are determined before a project or process starts. Instead, the dependencies between the quarcs in a project impose constraints that enforce an order in which quarcs can be started and closed. We argue that the entity-based QUARC process models are a superset of *activity-based* models, because there are at least two design alternatives for representing

activity-based models in a QUARC-based project. The first one is to define an enumerated attribute that represents the activity to which a quarc belongs, with enumeration values for each of the possible activities. The second design alternative is to represent each activity as a quarc and connect these activity quarcs with temporal dependencies as the modeled workflow requires.

Entity-based process models can more easily accommodate runtime process changes than activity-based models. A deviation from the standard flow in an activity-based process requires the process planner to foresee the potential deviation and make provisions for it, if the modeling language or notation he uses allows such constructs. For example, the *Business Process Model And Notation* (BPMN, [Obj11a]) defines Exception Flows ([Obj11a], p. 35) to deal with deviations that the BPMN process designer expects at design time. This approach is inadequate to handle *unexpected* deviations. The BPMN process designer can only design for deviations that can be anticipated, but can not model *unforeseen* circumstances. The QUARC metamodel can represent standardized processes, but allows the process model to deal with unpredictable events.

There are several ways to address this shortcoming by supporting runtime modifications of activity-based processes. Müller and Rahm [MR99] propose the ability to change medical treatment plans for patients at runtime when exceptions occur, such as a patient's unexpected intolerance of specific medication. They model a system based on predicates and rules which are defined at design time. At runtime, when a set of predicates evaluates to true, the associated rules declare how the workflow is altered. Sadiq et al provided a framework to specify incomplete processes that can be completed by using predefined building blocks at runtime ([SO98]). This approach constructs processes from a predefined pool of activities called *blocks*. The process cannot be changed at runtime to handle situations for which the pool contains no predefined block. A QUARC based process can accommodate arbitrary changes at runtime. There is no distinction between design time and runtime, tailoring is allowed throughout the entire lifecycle.

2.2 Quarc Origins: Issues and Rationale

The QUARC metamodel is an evolution of the IBIS issue concept introduced to support design [KR70]. IBIS was designed to address complex, ill-defined ("wicked") problems with multiple stakeholders. In IBIS, the core entity is the *Issue*, which represents a problem or decision that needs to be resolved. In a project, there are interconnected issues, *Position*s that present possible solutions for an issue, and *Argument*s that support a position or object to it. One of the goals of IBIS is to capture decision rationale, i.e. the reasons behind a decision, which is recorded in Arguments in IBIS. Conklin and Begemann implemented IBIS as a hypertext-based client/server tool named gIBIS [CB88]. There have been several approaches to evolve the IBIS concept in the past, for example Questions/Options/Criteria by MacLean et al. ([MYM89] [MYBM91]). QOC shows the design space around an artifact with a notation based on *Questions*, *Options*, and *Criteria*. Questions identify key design issues, Options provide possible answers to the Questions, and Criteria serve to assess and compare the Options. Arguments discuss the correctness of assessments of Options with respect to Criteria, and Arguments can also discuss Arguments. Dutoit generalized the QOC concept in his ReQuest approach [DP02].

IBIS and the other mentioned approaches all focus on design rationale. The QUARC approach has a wider reach, it aims to connect design rationale with task management, risk management and software lifecycle modeling.

Wolf attempted to unify system modeling, rationale management, and task management in RAT [WD04] and Sysiphus [Wol07] [BDW06]. Helming and Koegel extended Sysiphus and generalized the idea of requirements traceability [GF94] to include traceability between tasks and system model elements in UNICASE [BCH07].

In the QUARC metamodel, we generalize these representations and include communication other than plain text, e.g. voice and video. Our framework supports a formal approach to brainstorming without imposing a rhetorical model. One of the earlier approaches to capture informal communication is iBistro ([BBD01], [BDHB02]), which focuses on text and requires extensive post processing of the captured information. Mind-mapping tools allow the creation of new ideas without a rhetorical model, but there is no process for the transition of brainstorming bubbles to objects or use cases, they focus on the structuring of issues. In the QUARC approach, the bubbles can be represented directly as quarcs, which can be refined later.

2.3 Concepts

2.3.1 Quarcs

The QUARC metamodel defines just one single entity, the *Quarc*.

Definition 1 (*Quarc*)

> *A quarc represents a problem for which potential solutions must be found and decided on, and/or a work item to be completed.*

The decision and work item aspects are independent from each other. Decisions and work items have no fundamental differences, they are different points of view on the same basic project entity from different perspectives.

2.3.2 Quarc Attributes

One of the goals of the QUARC metamodel is to facilitate automated operations on the project model. In order to allow such operations on quarcs, they must exhibit characteristics that a machine can understand. For humans, it would be sufficient to attach a picture of the text "this is done" on a napkin to indicate that a quarc is closed. A machine cannot understand this with reasonable effort. Therefore, we introduce *attributes* for quarcs to enable processing by machine.

Definition 2 (*Quarc Attribute*)

> *A quarc attribute is a named attribute on quarcs, similar to an instance variable on a class. A Quarc attribute is either enumerated, i.e. an attribute that declares a finite number of different possible values, numeric, or a date.*

A quarc attribute definition is independent of any specific quarcs. We believe these three types (enumerated, numeric, date) to be sufficient, but the basic concept can be extended to cover other types, such as strings. Attributes can be used to categorize quarcs along an arbitrary number of axes. Figure 2.1 shows typical quarcs from the areas that the QUARC metamodel applies to: issue management, task management, risk management and software lifecycle management. Note that the figure shows five instances of the same class, but with different attributes. This is technically incorrect in a UML diagram, but accurate for the QUARC metamodel because quarc attributes are dynamic. We present a specialized notation for dynamic quarc attributes in section 2.6.2.

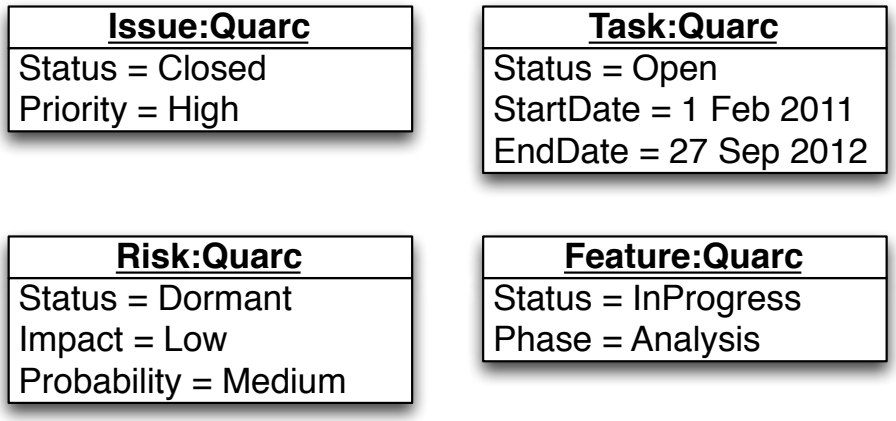


Figure 2.1: Quarc attributes illustration

We define four default attribute for quarcs: Status, StartDate, EndDate, and Duration.

Definition 3 (*Status*)

Status is an enumerated quarc attribute with the values Open, InProgress, Closed, Dormant. It represents the status of the task, decision, or other entity that the quarc represents.

Definition 4 (*StartDate*)

StartDate is a date-type quarc attribute that contains the start date of the quarc for task management purposes.

Definition 5 (*EndDate*)

EndDate is a date-type quarc attribute that contains the end date of the quarc for task management purposes.

Definition 6 (*Duration*)

Duration is a numeric quarc attribute that contains the work duration of the quarc for task management purposes.

See figure 2.2 for typical transitions of the Status attribute.

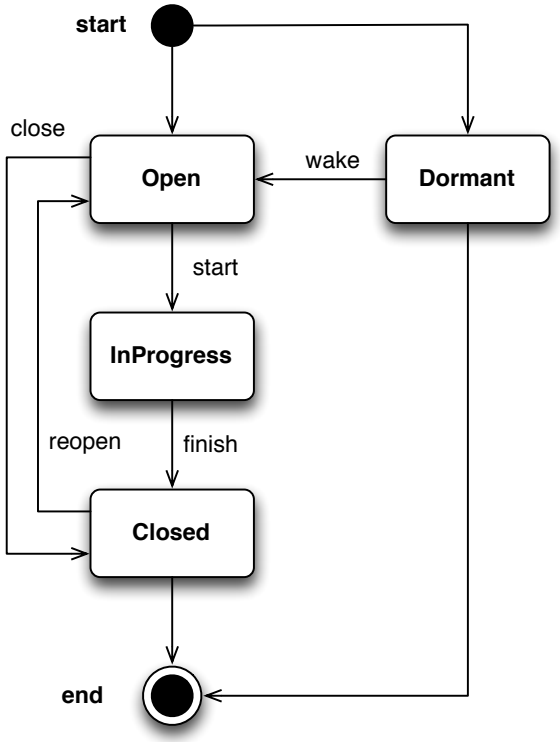


Figure 2.2: Typical quarc status transitions

2.3.3 Quarc Relations

Quarcs are connected by quarc relations.

Definition 7 (*Quarc relation*)

A quarc relation is a typed, directed connection between two quarcs.

Definition 8 (*Quarc graph*)

A quarc graph is a visual representation of quarcs and quarc relations in the form of a graph, where quarcs are represented as nodes and quarc relations as multi-colored directed edges (each quarc relation type has a unique color).

Figure 2.3 shows a quarc graph with the four quarcs from fig. 2.1, connected by three relations with two different types, indicated by the line pattern.

We define three default quarc relation types: startDependsOn, closeDependsOn, and alternativeCloseDependsOn.

Definition 9 (*startDependsOn*)

startDependsOn is a quarc relation type. A startDependsOn quarc relation from quarc Q1 to Q2 mandates that the status of Q1 may not be InProgress or Closed unless the status of Q2 is Closed or Dormant.

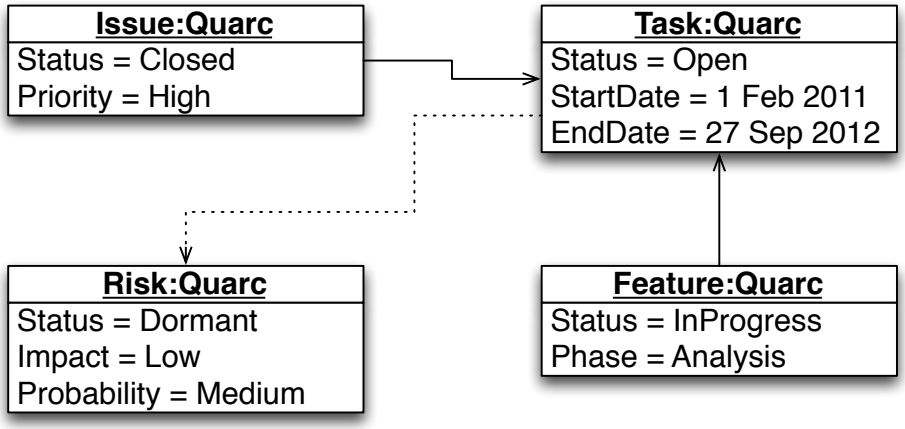


Figure 2.3: Quarc graph illustration with two different quarc relation types

Definition 10 (*closeDependsOn*)

closeDependsOn is a quarc relation type. A closeDependsOn quarc relation from quarc Q1 to Q2 mandates that the status of Q1 may not be Closed unless the status of Q2 is Closed or Dormant.

Definition 11 (*alternativeCloseDependsOn*)

alternativeCloseDependsOn is a quarc relation. When quarc Q1 has alternativeCloseDependsOn quarc relations to the set of quarcs {Q2}, the status of Q1 may not be Closed unless the status of all members of {Q2} is Dormant, or at least one member of {Q2} has the status Closed.

The startDependsOn quarc relation is typically used when the result of a task represented by the relation owner quarc is required as input to the task that the relation target quarc represents, i.e. when work on one quarc can only start after the work on another is finished. closeDependsOn quarc relations, on the other hand, are usually employed to make a group of quarcs part of another quarc insofar as that this quarc cannot be closed before all quarcs from the group are closed. This indicates a hierarchical parent-child dependency. An example for this is a quarc that represents the task to build a software subsystem. This quarc has closeDependsOn quarc relations to all quarcs that represent tasks required to build the subsystem.

The third quarc relation alternativeCloseDependsOn expresses an alternative choice. Before a quarc can be closed, *any* of the quarcs to which it has an alternativeCloseDependsOn quarc relation must be closed. This is different from the closeDependsOn quarc relation, because *all* of the quarcs to which a quarc has a closeDependsOn quarc relation must be closed before the quarc itself can be closed. alternativeCloseDependsOn is designed to connect a quarc representing a task to other quarcs that represent different solution alternatives. Figure 2.4 shows a quarc subgraph that models an open decision with several alternatives. While the decision is in progress, neither the task nor the solution alternatives can switch to InProgress because of their startDependsOn quarc relations. When the decision is made, as shown in figure 2.5, the decision quarc switches to Closed, which allows the solution alternatives and the task itself to switch to InProgress. Of course, out of the solution alternatives, only the chosen solution quarc actually does switch.

Due to the alternativeCloseDependsOn quarc relations from task to solutions, the task cannot be closed before the opened solution is closed.

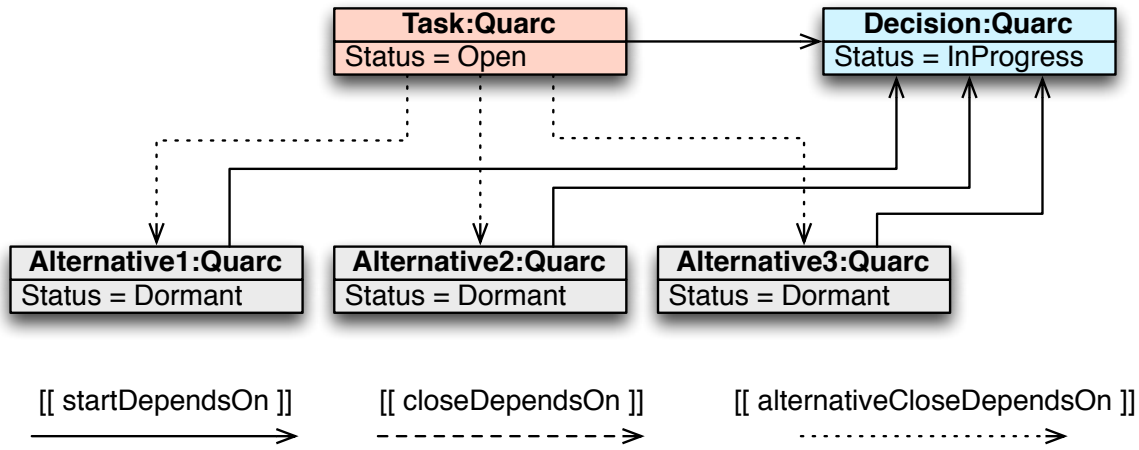


Figure 2.4: Example for alternativeCloseDependsOn quarc relations (decision in progress)
This diagram shows a task with three alternatives, and a decision that chooses between the alternatives. The decision is currently in progress, and the alternatives are all dormant. The task cannot start because the decision is not yet closed.

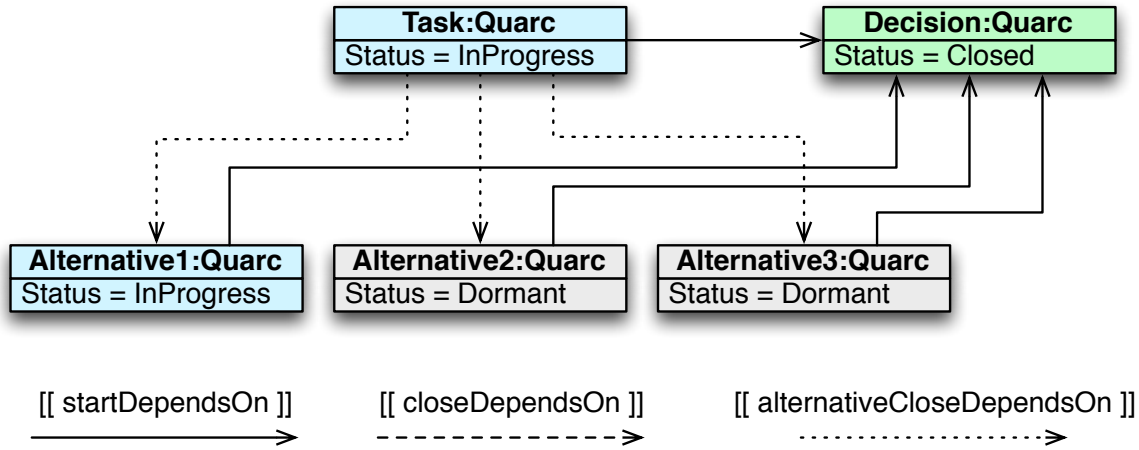


Figure 2.5: Example for alternativeCloseDependsOn relations (decision made)
After the decision is made, the decision quarc is closed and one of the alternatives is opened. The task can then start, along with the chosen alternative.

The closeDependsOn and alternativeCloseDependsOn quarc relations make it possible to represent AND/OR trees in a Quarc repository. An OR node connects to its successors through alternativeCloseDependsOn quarc relations, while an AND node uses closeDependsOn quarc rela-

tions. A quarc can combine traits of both AND and OR nodes, because **all** quarcs with closeDependsOn quarc relations and **one** of the quarcs with alternativeCloseDependsOn quarc relations must be closed to close the quarc itself. See figure 2.6 for an example.

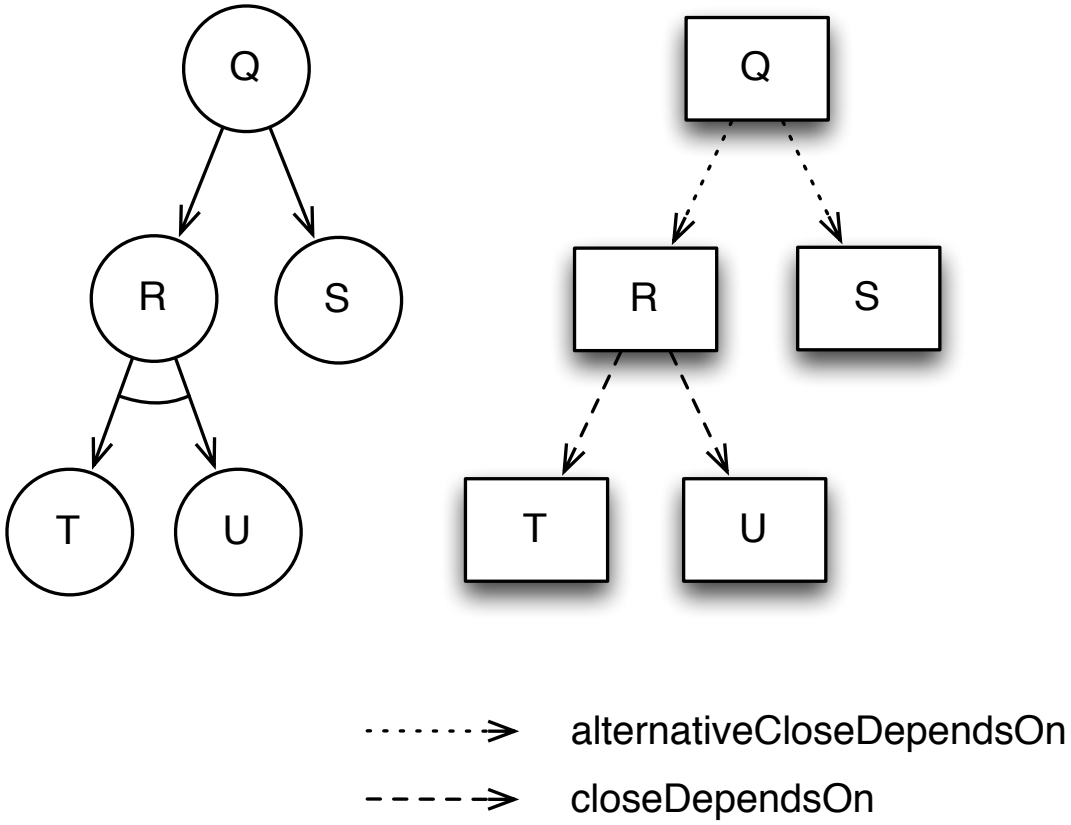


Figure 2.6: Quarc representation of an AND/OR tree

2.4 Quarcs and Communication

2.4.1 The QUARC Communication Model

The QUARC metamodel is designed to associate different kinds of project communication with quarcs.

Definition 12 (*Project Communication*)

> *Project communication is communication – such as meeting protocols, sketches, early drafts, informal models, or recordings – that has a relation to the project. It can be represented as text, photos, sound, video, drawings, or other media.*

Quarcs are intended to be discussed by project participants. The participants in such a discussion can record the conversation and connect it to the related quarcs. This kind of archived

information can be useful for many stakeholders, such as developers, knowledge managers, or project managers.

The QUARC metamodel conceptually supports any kind of media that can be digitized. It allows communication attached to quarcs to consist of just a voice recording, or other media (images, video, etc.). Therefore, our quarc model contains communication artifacts that can be read by human and machine (plain text), but it also contains communication artifacts that can (at this time) only be fully understood by humans (voice etc). A quarc repository may contain only machine-unreadable communication, but still be perfectly understandable by a human designer. A voice recording can even confer a model specification, such as a UML model (see subsection 2.4.4). In Appendix C, we present a small case study that finds audio recordings in a project management tool useful for project participants.

Definition 13 (*Communication Artifact*)

A communication artifact is one single media file that contains project communication.

Figure 2.7 shows the QUARC communication model.

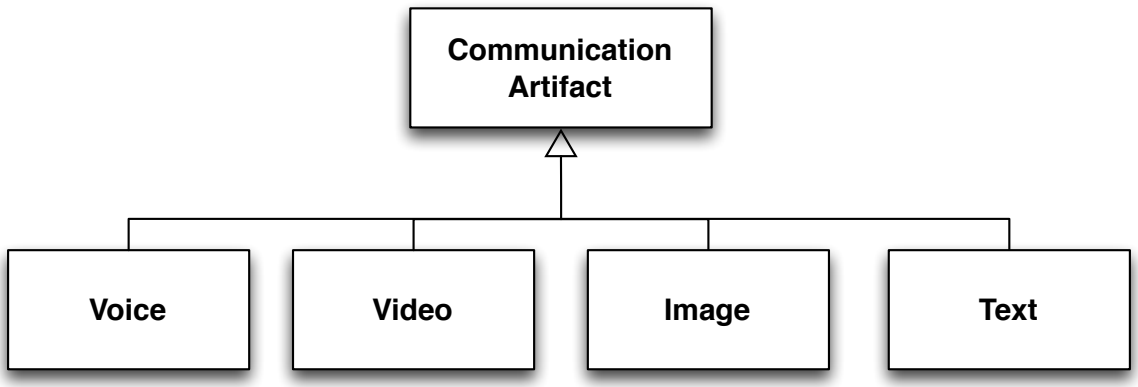


Figure 2.7: QUARC Communication Model

A communication artifact can contain several sequential or parallel discussions of different quarcs. In order to enable immediate access from a quarc to relevant parts of its associated discussion, a link from quarc to communication artifacts points to a *communication segment* instead of the artifact.

Definition 14 (*Communication Segment*)

A communication segment is a part of a communication artifact.

A communication segment of a voice recording is a time range of the communication artifact. A communication segment of a text document consists of a character range, i.e. a range starting at one character (identified by its position in the text) and ending at another. Figure 2.8 shows the relation between communication artifacts and communication segments in the QUARC metamodel.

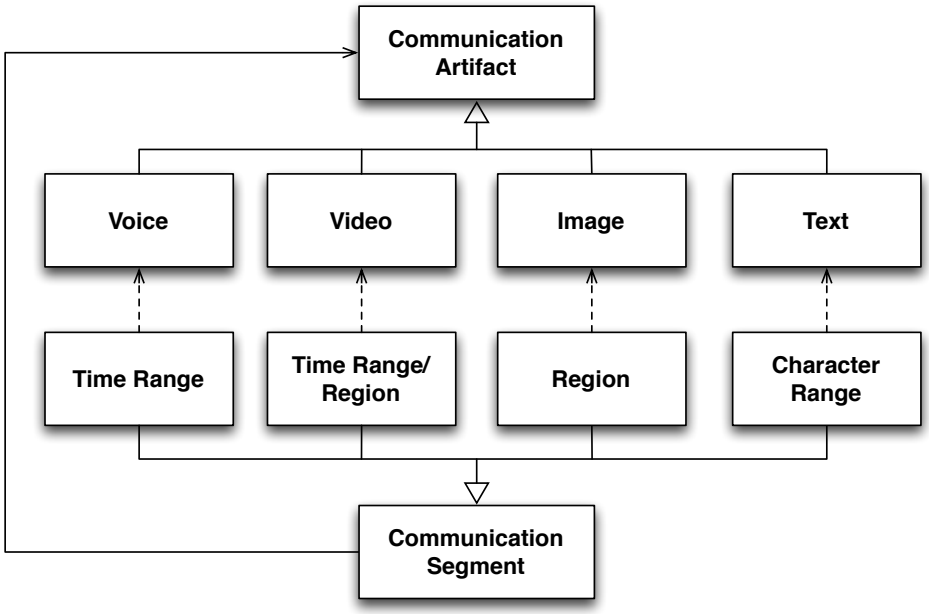


Figure 2.8: Communication Artifacts and Segments

Quarcs are linked to communication segments through *communication relations*.

Definition 15 (*Communication Relation*)

A communication relation is a typed relation from a quarc to a communication segment.

We define two default communication relations, describedBy and discussedBy.

Definition 16 (*describedBy*)

describedBy is a communication relation from quarc Q to communication segment C that indicates that C describes Q.

Definition 17 (*discussedBy*)

discussedBy is a communication relation from quarc Q to communication segment C that indicates that C discusses Q.

2.4.2 Informal Communication

Definition 18 (*Informal Communication*)

Informal communication is a project meeting that occurs spontaneously, without an agenda, usually in a mobile setting without installed recording infrastructure.

Modeling informal communication is an active area of research in computer/human interaction. Previous solutions for recording project communication do not cover informal communication due to the lack of ubiquitous availability of recording infrastructure. Another problem is

the difficulty to capture the context of informal communication, which makes it hard to find specific information in a large number of recordings. This has partially been solved, for instance by Wactlar and Hauptmann in the Informedia project. They employ speech recognition followed by natural language processing to provide access to relevant communication segments, in this case for TV news [CWH99].

We see two more reasons why informal project communication is generally neglected in formal process models. First, recording informal communication requires a capturing infrastructure that is expensive to install and maintain, if it were deployed in sufficient volume to cover most or all of the points where informal communication might occur. Second, to make captured communication discoverable in a project, it should be referenced or included in artifacts of a project management model. Due to the sequential nature of informal communication artifacts, i.e. voice and video recordings, it is difficult to quickly scan larger amounts of such material for specific information. Metadata such as a timestamp, location and maybe even participants may not be sufficient for fast discovery of communication segments that relate to a specific topic. In the QUARC metamodel, the annotation of video streams can be represented by communication relations between quarcs and communication segments of the video stream (communication artifact).

[1] Hindus et al say that it would be desirable to record all conversations if possible [HSH93]. They call this "ubiquitous audio", but do not have any suitable infrastructure at their disposal. They also rely on automated content analysis, which isn't yet available in a useable form.

Cherry et al recorded audio and video of team interactions to measure the impact of different participant types on team dynamics [CR09]. They did not link their recordings to formal models and required video cameras with microphones to be installed at all points where conversations occur that they wanted to record.

Tucker et al recognize the benefits of storing recorded speech, but organize and access audio on the recording device instead of linking it to model elements [THH03].

Stiefelhagen et al record meetings with omnidirectional cameras, but do not cover informal communication and formal model links [SCY05]. Lee et al propose a portable audio/video meeting recorder [LEG+02], although this equipment is still limited and not portable enough for spontaneous recordings.

Purver et al recorded meetings to analyze their content automatically. They did not cover informal and unplanned conversations, and focused on content analysis [PEN06], [PDN+07].

Ogata et al recorded meetings in audio form and used an automated process that detects laughter, coughs and pauses to identify the meetings' structure [OA06].

Basu et al instrumented a room with sensors to observe and influence the behavior of meeting participants [BCCP01]. This requires static infrastructure and is unsuitable for informal

[1] The remainder of this section was adapted from a paper by Nagel et al on audio recording in software engineering [NHKN10].

conversations in situations where those sensors are not present.

Richter et al presented a traceability framework to capture and access multimedia sources [RAG+01], [RGLL06] and showed that access to captured information is beneficial for requirements engineering processes. They analyzed video recordings of meetings and knowledge acquisition sessions and provided an index structure for traceability. However, they relied on complex and limited infrastructure (video recording equipment) and created manual and semi-automatic transcripts of the recordings in order to index them. This process requires manual processing and does not scale well.

Sawhney et al propose an infrastructure for asynchronous audio communication, but do not link their audio messages to formal context elements [SS00].

Fahmi et al researched the use of mobile devices for requirements engineering [FIC07]. They used web-based requirements elicitation tools without audio recording capabilities on PDAs. This means that participants had to type all information that they deemed important into the PDAs. Typing creates higher friction for information acquisition than an audio recording which doesn't require interaction from the participants except for the initial action to start recording.

2.4.3 Meeting Management

In structured communication environments such as scheduled meetings, infrastructure required for capturing communication can be deployed. There are many approaches to meeting management that rely on capturing communication. Some intend to preserve it as rationale [DMMP06], others try to process it automatically to extract information that is relevant for project management. Automated processing often relies on artificial markers in the voice stream (reserved signal words or phrases) or on heuristic approaches.

Schiller [Sch11] uses word spotting to identify action items and other project management entities. Her approach requires all participants to bracket these entities in reserved signal words/phrases (e.g. "Begin Action Item" ... "End Action Item"). The meeting participants have to remember to use the reserved phrases correctly and structure their conversation into entities that the word spotter understands. Both these requirements are hard to fulfill in a meeting that involves creative tasks and design aspects, because they impose a rhetorical model on the participants that may inhibit the free flow of ideas.

When a meeting is being captured for rationale purposes, Schneider argues that the burden of structuring the captured media cannot be placed on those that generate the rationale during the meeting [Sch06]. He proposes that a person not involved in the meeting should perform the structuring. If the communication is captured in a quarc repository, the structuring task is much simpler than in traditional models such as IBIS, because an initial quarc representation can be unorganized and unstructured, to be refined later. For example, there is no requirement that a quarc represents only one question as in IBIS. A rationale capturer has to structure an ongoing meeting conversation into a mental model that matches the concepts of the used ratio-

nale model. Next, he has to externalize the mental model into the destination medium (e.g. a rationale management system).

In IBIS, the rationale capturer has to consider for each sentence whether it is an issue, a proposal, a criterion, a pro or a contra, and create an appropriate entity during externalization. This amount of mental transformation and structuring is hard to perform on-the-fly, while a meeting is in progress. In a QUARC environment, the capturer can simply create a few quarcs that represent the general topics of discussions and link communication segments during a meeting. Later, he can refine the quarcs and communication segments. The flexibility of the QUARC metamodel places very few restraints on how the capturer externalizes his mental model.

2.4.4 The Use of Abstractions in Communication Segments

Christopher Alexander, in his book on architecture, has already put forward the idea that design concepts can be expressed in communication as references to design patterns [AISJ77].

> ROOMS TO RENT... this pattern is the first which sets the framework for the outbuildings. Used properly, it can help to create NECKLACE OF COMMUNITY PROJECTS (45), THE FAMILY (75), SELF-GOVERNING WORKSHOPS AND OFFICES (80), SMALL SERVICES WITHOUT RED TAPE (81), FLEXIBLE OFFICE SPACE (146), TEENAGER'S COTTAGE (154), OLD AGE COTTAGE (155), HOME WORKSHOP (157): in general it makes any building flexible, useful in a greater variety of circumstances.

In design sessions in early phases of a software project, system models are often created ad-hoc and discarded quickly. These models can be inconsistent, incomplete and on any level of abstraction, as long as the involved parties understand their content. For a discussion among humans, a rough sketch of an idea on a napkin can be enough to express an idea. It is often sufficient to involve fragments of models in a human-to-human conversation. An example for such model fragments would be design patterns.

Consider the following scenario: Two developers are planning the software architecture of their current project. At one point, they're debating whether they should introduce an Observer pattern [GHJV96] at a specific point in their architecture. Both developers are not familiar enough with their system architecture model to know the objects that might participate in the Observer pattern, so they use a paper printout of their architecture during their discussion. The physical representation of their architecture is referenced in their communication. The printout is an abstraction that shows objects of their UML model without attributes or methods. This abstraction level doesn't contain all details about the model, but it is sufficient for the current communication. The Observer pattern, on the other hand, is a commonly known model fragment for both parties. They are even familiar with its attributes and methods, which is beyond what's required for the conversation. During the conversation, one of them determines that they need to refactor one object in their architecture into two different ones. He describes the two new objects to his colleague. This description represents a model fragment embedded in their communication.

Model fragments, if discussed by reference, might not be available to all parties involved in the conversation on the required abstraction level. For example, if two people discuss specific methods of the Observer pattern, but one of them only knows the involved objects, but not their methods, then the abstraction level of his mental representation of the Observer pattern model fragment is too high for the current discussion. He will have to switch to an actual representation (e.g. paper printout) of the Observer pattern on a lower abstraction level. If communication participants use different representations of a model fragment, it is possible that their representations do not match exactly. This may lead to misunderstandings that take time to resolve. Mental representations are especially prone to mismatches. When it is not possible to use model fragment references in a communication due to availability or mismatch problems, the involved parties may try to switch to using model fragments embedded in the conversation.

2.5 Quarc Repository

Quarcs are organized in a container, the *Quarc Repository*. A quarc repository represents a logical context for all the quarcs it contains, for example a single project.

Definition 19 (*Quarc Repository*)

> *A quarc repository is a container for quarcs, quarc attribute definitions, quarc relation type definitions, communication relation type definitions, and communication segments.*

These definitions form a model for the information represented by attributes and relations, and therefore establish a structure on the quarcs and communication segments. Communication artifacts are not explicitly represented in the QUARC metamodel. The QUARC metamodel treats all media as binary black boxes, which are linked to quarcs through CommunicationSegments.

A quarc repository enforces structural integrity constraints. An example for such an integrity constraint is the startDependsOn quarc relation. From a structural point of view, this relation is just a relation between quarcs. Its meaning, though, is that not all of the possible Status attribute value combinations on the two ends of the relation are allowed. The combination of InProgress on the origin quarc and Open on the destination quarc is prohibited, whereas InProgress/Closed is permitted (see definition 9). Integrity constraints are formulated as programs assigned to a quarc repository, which the repository runs after each transaction on its content. We call these programs validators, because they check if the structure of a repository is valid. In appendix B, we show validators for the default quarc relations startDependsOn, closeDependsOn, and alternativeCloseDependsOn.

2.6 The QUARC metamodel

The QUARC metamodel hierarchy is based on the UML meta model hierarchy [Obj11b] [Int07]. Figure 2.9) shows the six metamodel levels labeled Q0 to Q5. There are three levels of abstraction in the QUARC metamodel that would all occupy the M1 level in the UML meta model. First is the QUARC model, which defines quarcs, quarc repositories, attributes, quarc relations etc., on level Q3. Second is the quarc repository model level, which contains partially

instantiated quarc repositories that serve as models for different model representations, on level Q2. Finally, on level Q1, are fully instantiated quarc repositories that represent concrete models. The highest level, Q5, is the OMG's Meta-Object Facility (MOF).

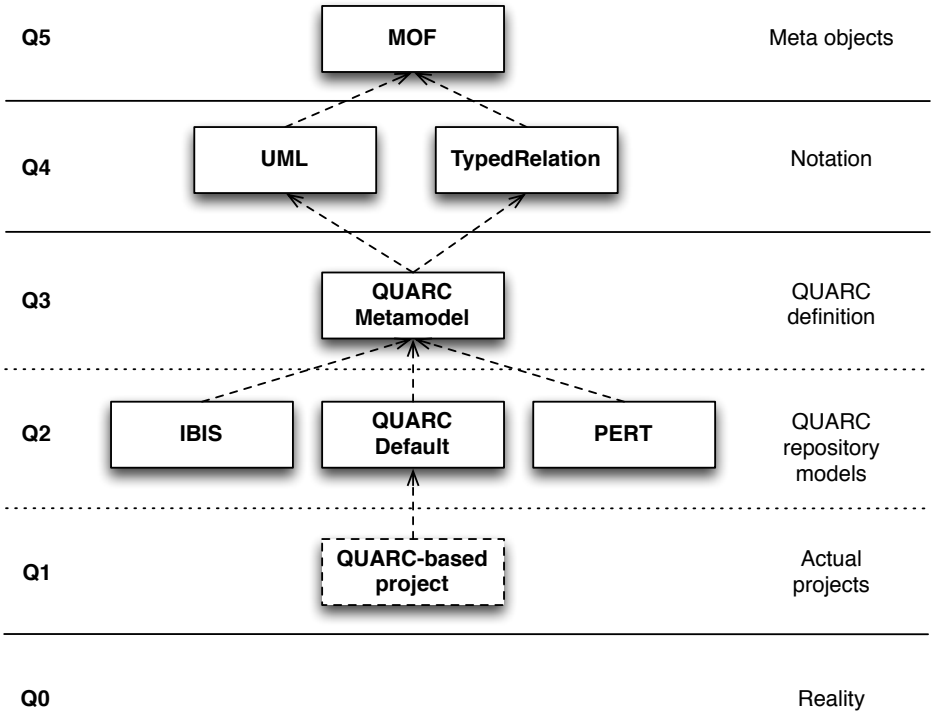


Figure 2.9: The QUARC metamodel hierarchy levels

2.6.1 Level Q4

Level Q4 defines a ternary association between classes which we call *Typed Relation*.

Definition 20 (*Typed relation*)

A typed relation is a directed association from a relation owner to a relation target that has a relation type.

Definition 21 (*Relation class*)

A relation class is a class that defines the type of a typed relation.

See figure 2.10 for a visualization in UML notation. To make diagrams that contain multiple typed relations more readable, we define a new notation for typed relations as a line from relation owner to relation target, with an open arrow on the target's side and a "crow foot" pattern at the owner's side. At one point on the line, a second line is attached orthogonally to the first line. It ends at the relation type class. We also define an abbreviated notation to further improve readability and clarity of diagrams. It only consists of the line between relation owner and target, and a label. The label is enclosed in double square brackets and shows the relation type class

name. It can also show an instance of a relation type class (in UML notation, Instance:Class).

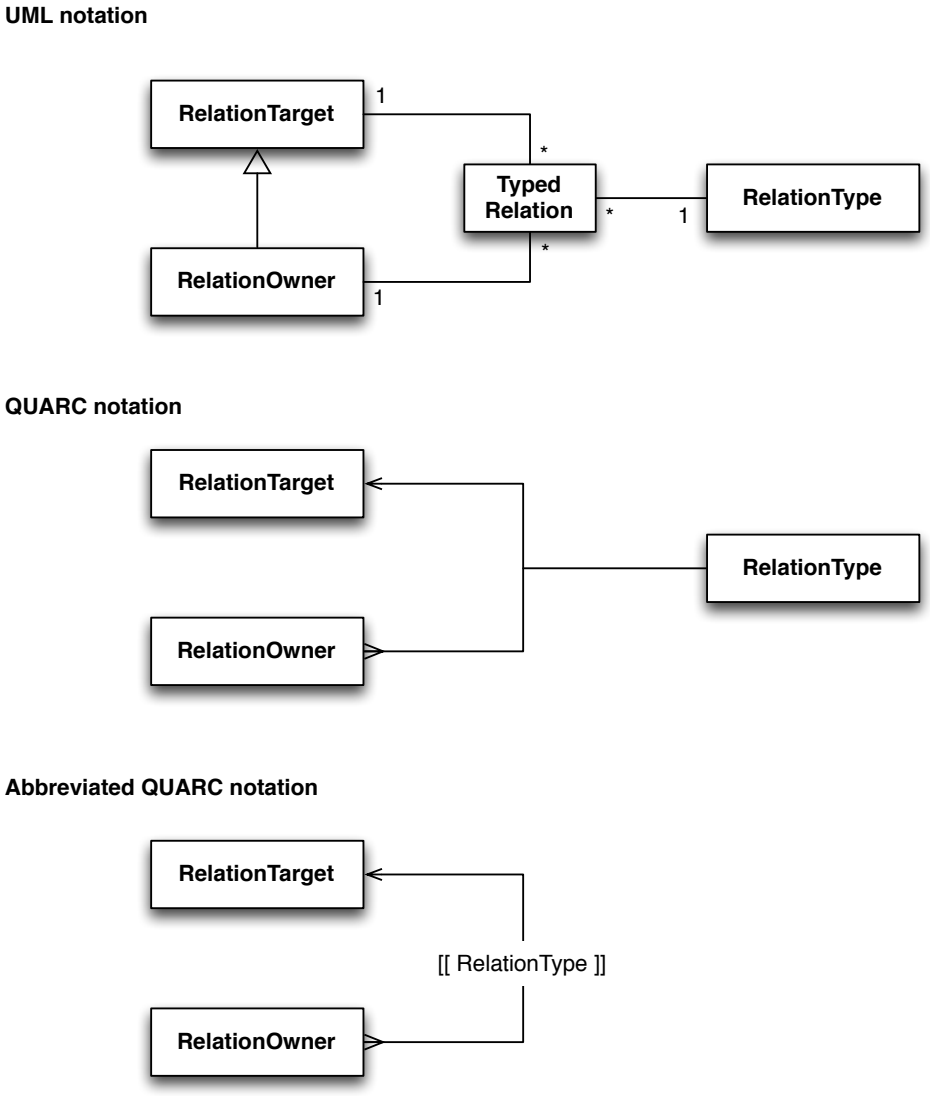


Figure 2.10: *Typed Relation* **notation**

2.6.2 Level Q3

Level Q3 defines the QUARC model. It consists of the QuarcRepository, Quarcs, QuarcRelations, Attributes, AttributeValues, CommunicationSegments, and CommunicationRelations. The UML class diagram in figure 2.11 shows that the quarc repository consists of type definitions for quarc relations, attributes, and communication relations, as well as quarcs and attribute values. The quarc repository defines methods to alter these type definitions and to manage quarcs. The diagram also shows annotation links to model elements of a system model (ModelElement), and links to placeholders in a system model (ModelPlaceholder). The placeholder concept will be

explored further in section 2.7.

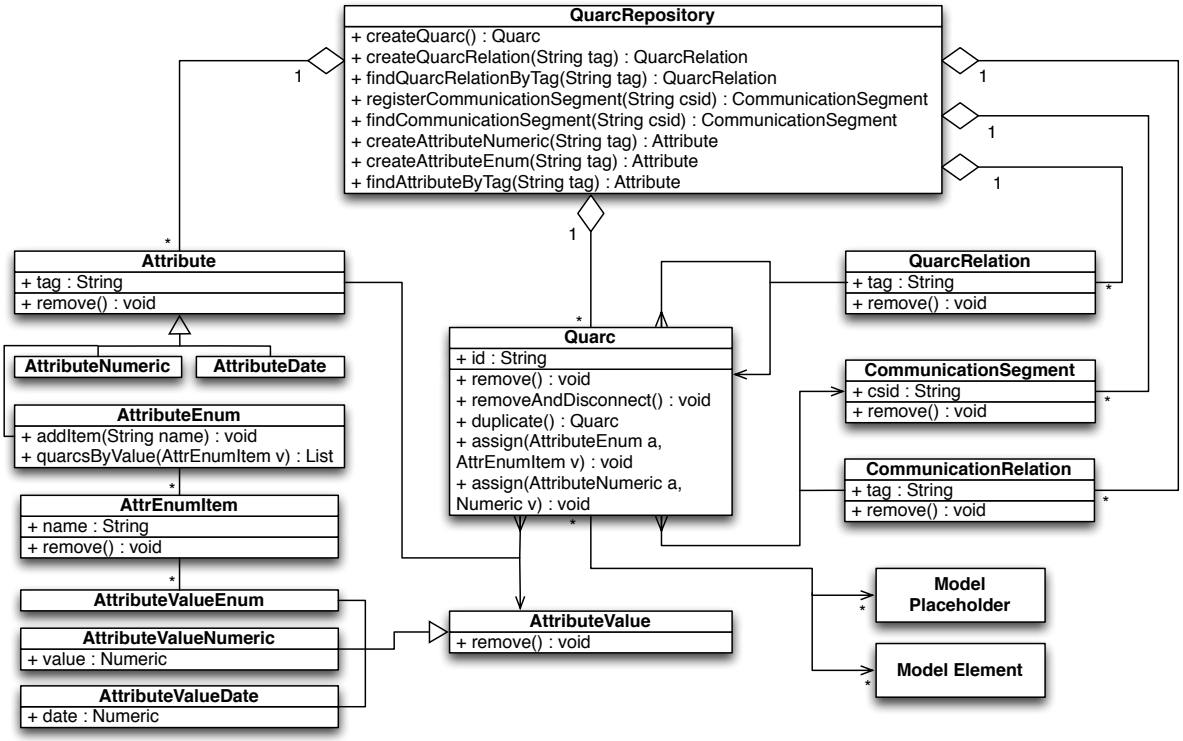


Figure 2.11: The Q3 QUARC metamodel

The QUARC model defines attributes as separate classes to support modification, i.e. adding and deleting attributes, at runtime, to allow dynamic tailoring. The Attribute class is the superclass of all attribute type classes. Attribute types are identified by a tag. Numeric attribute types are represented by the AttributeNumeric class, date attribute types by the AttributeDate class. Enumerated attribute types are represented by the AttributeEnum class, individual enumeration elements by the AttrEnumItem class. For each attribute type, there is a class that models a concrete value. AttributeValue is the superclass of these attribute value classes. AttributeValueNumeric, AttributeValueDate, and AttributeValueEnum represent values for AttributeNumeric, AttributeDate, and AttributeEnum, respectively. The typed relation between Quarc and AttributeValue, type Attribute, represents the assignment of attribute values for a specific attribute type to a quarc.

Quarc relations are modeled as typed relations (see subsect. 2.6.1) in the QUARC model. The relation class for quarc relations is QuarcRelation. Communication relations are also typed relations in the QUARC model; their relation class is CommunicationRelation.

The QUARC model defines a shorthand notation to simplify the notation of quarc attributes in Quarc classes and instances. In UML notation, attributes are static, i.e. all instances of a class have the UML attributes of the class. Attributes in the QUARC model are dynamic: two instances of the Quarc class can have different attributes assigned to them, and the set of defined

attributes can change during runtime of a project. Both aspects cannot be represented by the UML notations, so the QUARC model defines a new notation. A Quarc - Attribute - AttributeValue typed relation does not involve other quarcs, so the attribute is noted in a compartment of the Quarc class or instance. To show that there is an attribute named AttrName which is required to be set on a Quarc, its compartment contains an entry of the form [[AttrName]]. For enumeration attributes, the entry may show the list of enumeration items as a comma-separated list enclosed in curly braces, e.g. [[AttrName]] : { value1, value2, ... }. To express that a concrete attribute value is assigned to a Quarc instance, its value is written after the attribute name, separated by an equal sign, e.g. [[AttrName]] = value. Figure 2.12 shows a quarc class with an attribute definition that lists all possible values, and a quarc instance with a concrete attribute value assignment.

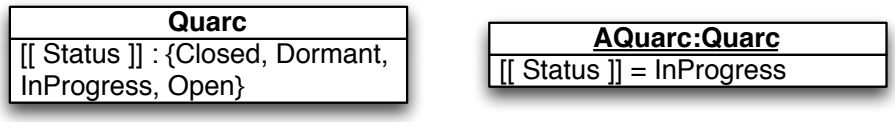


Figure 2.12: Quarc attribute notations
The Quarc class on the left defines an enumerated attribute Status with possible values. On the right is a Quarc class instance with attribute Status set to InProgress.

The notation for concrete attribute values can not only be applied to Quarc class instances, but also to Quarc classes. When a specific attribute value is assigned to an attribute in a Quarc *class* in a diagram, then that class represents the *set of all* Quarc instances that have this attribute value set for the attribute. As a consequence of this notation, the Quarc *class* may appear several times in the same diagram, if all appearances have disjoint attribute assignments. If there are Quarc classes with attribute assignments in a diagram, then there may be an additional Quarc class without any attribute assignments that represents all instances that do not match any of the other classes' attribute assignments. Figure 2.13 shows a diagram with multiple Quarc classes. The class on the left represents all instances that have the value Closed assigned for the Status attribute. The class in the middle represents all instances with InProgress assigned for Status, and the class on the right represents all other Quarc class instances.

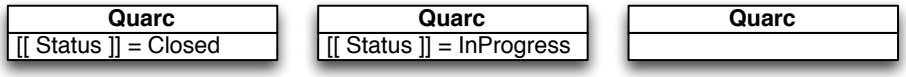


Figure 2.13: Multiple Quarc classes in the same diagram
This diagram shows three sets of Quarc instances, expressed by three Quarc classes with different attribute assignments.

2.6.3 Level Q2

Level Q2 of the QUARC metamodel deals with partial instantiations of the QUARC metamodel. We call these *quarc repository models*.

Definition 22 (*Quarc Repository Model*)

> *A diagram that consists of classes from the QUARC model (Q3), and instances of these classes.*

A quarc repository model defines instances of the quarc relation types and communication relations, as well as attribute type instances. The default QUARC repository model (shown in figure 2.14) models the default attributes and relations from subsections 2.3.2, 2.3.3, and 2.4.1. Figure 2.14 and others that depict quarc repositories on level Q2 contain not only classes, but also instances. This kind of mixture illustrates the concept of predefined attributes and relations in a more easily accessible way than separate class and instance diagrams. The included instances in the default quarc repository model represent the default attribute Status, the default quarc relations startDependsOn, closeDependsOn, and alternativeCloseDependsOn, as well as the default communication relations describedBy and discussedBy. In the remainder of this dissertation, we assume these defaults to be present. Figure 2.15 shows figure 2.14 in the shorthand notation introduced in the previous subsection.

In chapter 4, we show how models for issue management (IBIS), task management (PERT) as well as risk management can be represented as quarc repository models.

2.6.4 Levels Q1 and Q0

Level Q1 models are fully instantiated quarc repository models, i.e. they don't contain any classes, only instances. Level Q0 is the real world, just like M0 in the UML metamodel hierarchy.

2.7 Quarc Repositories and System Models

Helming and Koegel generalized the idea of requirements traceability [GF94] to include traceability between tasks and system model elements in UNICASE [BCH07]. The QUARC metamodel also allows for traceability links from quarcs to system model elements. It extends this idea to include links from quarcs to *system model element placeholders*.

Definition 23 (*System Model Element Placeholder*)

> *A system model element placeholder is a nonstandard element in a system model that serves as a placeholder for an actual system model element. It is used to indicate that there needs to be a system model element (or a partial system model) in its place, but the system model designers do not yet know what to replace it with.*

For UML diagrams, we use a cloud symbol to represent a system model element placeholder. We chose the cloud symbol because is not part of the UML notation, and its shape is different enough from the UML notation elements to prevent it from being confused with an

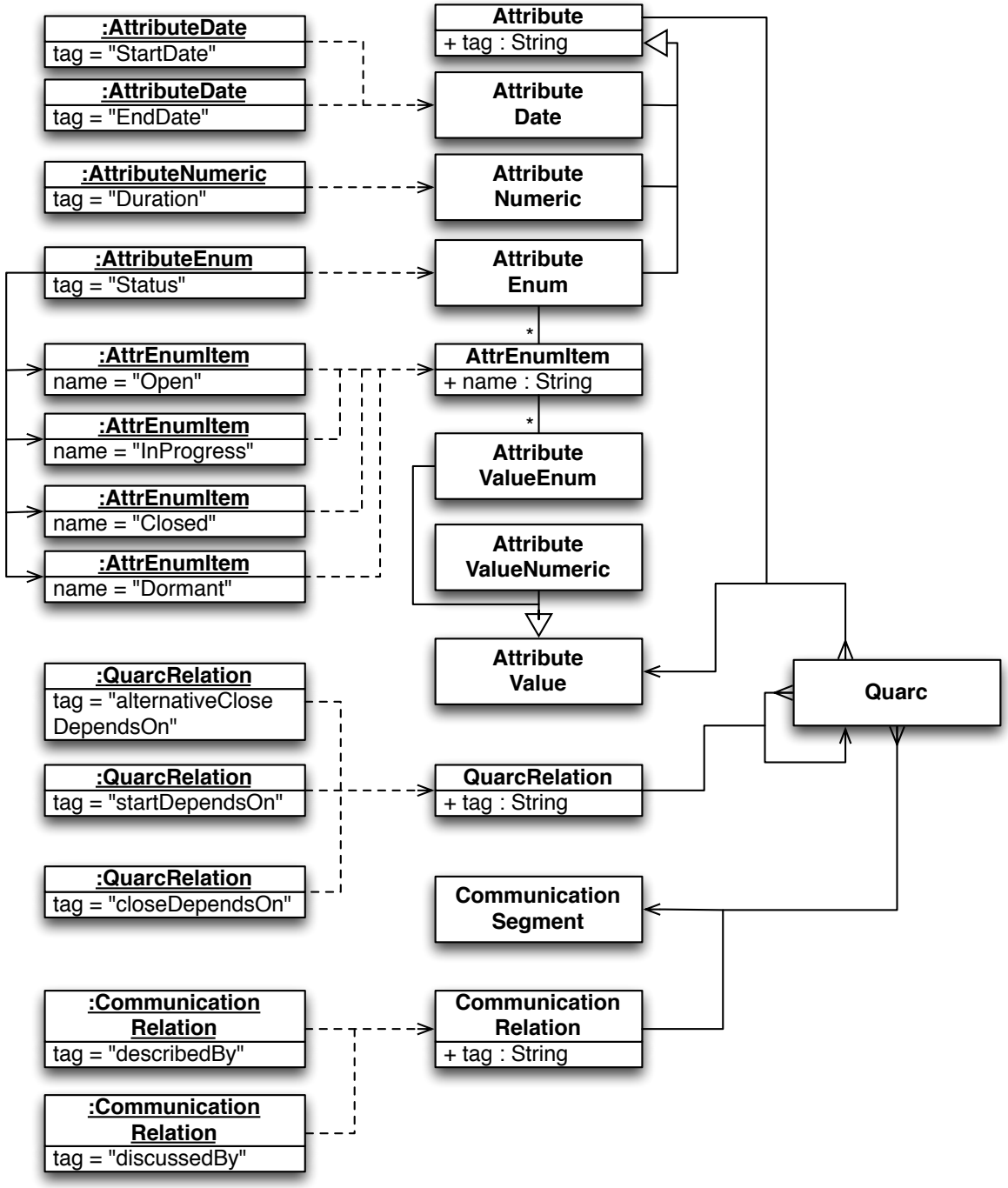


Figure 2.14: The Q2 default quarc repository
(methods and most attributes omitted for readability)

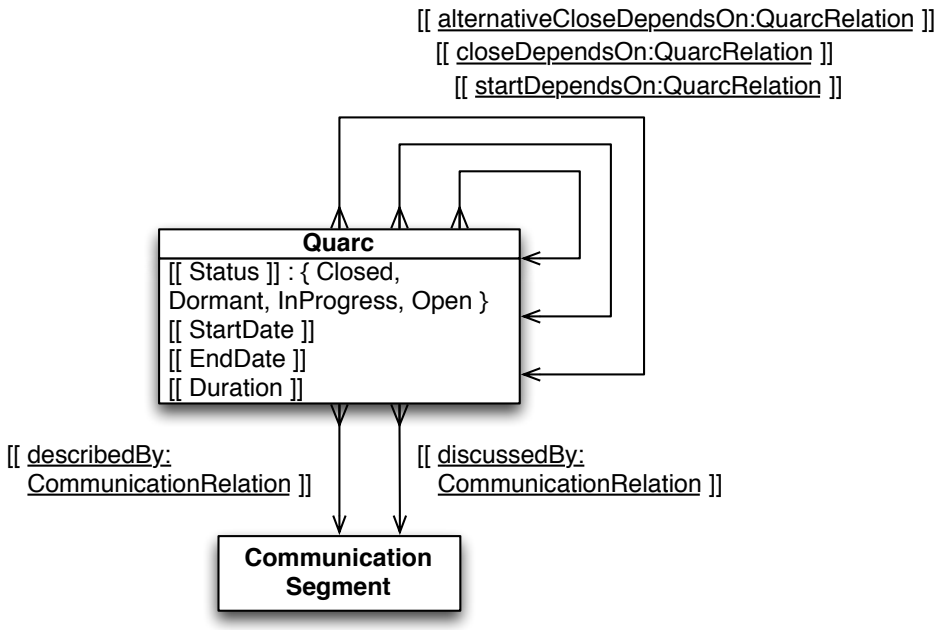


Figure 2.15: The Q2 default quarc repository (shorthand notation)

actual UML model element.

In the first example scenario, a software project's task is to build an application that collects and displays sensor data. The developers already identified three subsystems: a sensor data acquisition subsystem, a GUI subsystem, and a data processing subsystem to run filters, aggregations and statistical functions on the incoming data. They also know that there needs to be some kind of persistence subsystem, but still need to make a buy-vs-build decision. They identified three possible solutions: build a custom persistence subsystem, purchase an object-relational database (ORDBMS), or use a combination of open-source components, i.e. a database (DBMS) and an object/relational mapping layer framework. Figure 2.16 shows the system model for this project with the DataAcquisition, GUI and DataProcessing subsystems on the left. On the right side, there is a quarc graph that represents the decision for one of the three persistence subsystem alternatives. The Persistence quarc represents the task to build a persistence subsystem, the WhichPersistence quarc represents the decision process itself, and the three options are represented by the quarcs BuildOwn, UseOSComp, and BuyORDBMS. Linked to each option is a system model element representing the respective subsystem.

Figure 2.17 shows the situation after it was decided to use open-source components. The system model elements that were linked to the UseOSComp quarc have replaced the placeholder, and the quarcs have changed their status.

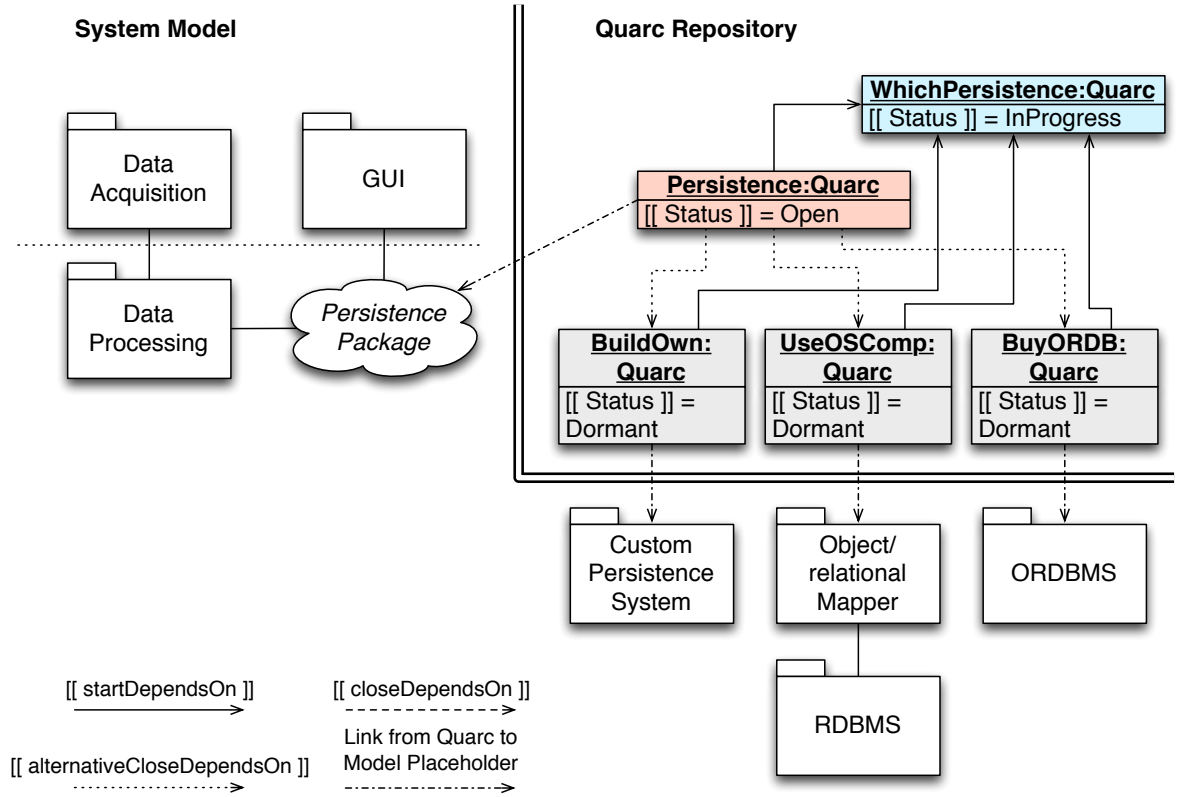


Figure 2.16: Placeholder in a package diagram

This diagram shows a part of a quarc repository that represents for the persistence subsystem of a software architecture. The task is connected to a placeholder for the persistence subsystem in a UML package diagram. Each solution alternative is connected to a replacement package for the placeholder.

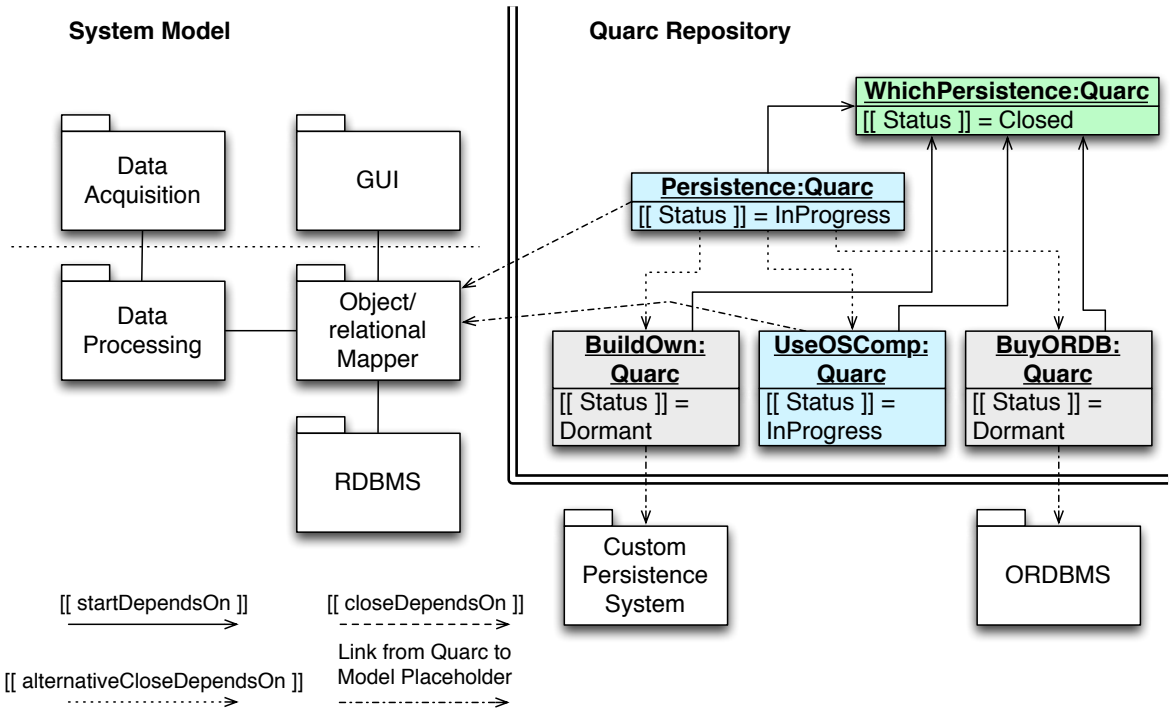


Figure 2.17: Placeholder substituted

The placeholder in the package diagram was replaced by the "Object/Relational Mapper" and "RDBMS" packages.

CHAPTER 3

The Quarc Query Language

We define a concise statically-typed procedural language, the *Quarc Query Language* (QQL), to access, modify and validate the content of quarc repositories at runtime. We had several design goals for QQL. It should be robust and secure, platform-independent and architecture-neutral. It should be able to express nontrivial programs such as graph searches, and allow all operations in a quarc repository that are described in chapter 2. In this chapter, we describe the structure of QQL.

3.1 Top Level Constructs

This section describes the syntax of QQL. All definitions in this section are given in EBNF form. The QQL grammar is a LALR(1) type grammar to allow it to be processed by the lexer/-parser generators *yacc* and *bison*. See appendix A for the complete grammar and its representation as *bison* source code. QQL is case-insensitive for keywords, variables, attribute names and tag values.

```
qql ::= ( transaction | validatordefinition | functiondefinition |
   variabledeclaration )+
```

Listing 3.1: QQL top level definition

3.1.1 Transactions

QQL statements that perform operations on a quarc repository are executed in the context of a *transaction*. Similar to an SQL transaction, a QQL transaction will either succeed completely, or make no changes to the quarc repository at all. Each QQL transaction has three phases: *execution*, *validation*, and *persisting*. During its execution phase, a transaction operates on an isolated transient copy of the repository, its operations are not yet performed on the actual repository. In the validation phase, the structural integrity constraints of the repository are validated (see next subsection). If validation fails, the transaction is aborted and the quarc repository remains unchanged. Otherwise, the changes made to the transient copy are recorded

in the quarc repository in the last phase.

```
transaction ::= 'transaction' statementblock
```

Listing 3.2: QQL transaction

3.1.2 Validators

In section 2.5, we introduced structural integrity constraints in the context of a quarc repository. These constraints are expressed as QQL *validators*. A validator is a QQL program that gets executed as a part of the validation phase of each transaction (see previous subsection). QQL validators have a purpose similar to referential integrity mechanisms ([Sto75], [EC75]) in relational databases.

```
validatordefinition ::= 'validator' validatorname statementblock
validatorname ::= identifier
```

Listing 3.3: QQL validator

3.1.3 Statements

QQL consists of *statements*. A statement is either a variable declaration, an assignment, a while loop, a for enumeration, an if conditional branch, a function call or a return. Return statements are only valid in the statement block of functions, not in statement blocks of transactions or validators. A sequence of statements enclosed in curly braces is a *statement block*.

```
statementblock ::= '{' statement ( ';' statement )* '}'
statement ::= variabledeclaration | assignment | while | for | if |
   functioncall | return
assignment ::= variable '=' expression
expression ::= literal | variable | functioncall | '!' '(' expression
    ')' | '(' expression operator expression ')'
operator ::= equalop | booleanop | compareop
equalop ::= '==' | '!='
booleanop ::= '||' | '&&'
compareop ::= '<' | '<=' | '>' | '>='
```

Listing 3.4: QQL statements

3.1.4 Variables and Entity References

QQL variables are declared within a statement block prior to their first use in a statement. QQL variables are global, if they are declared outside of all statement blocks. Variables are scoped, i.e. they are only visible within the statement block where they are declared after the point of their declaration (except global variables, which are visible everywhere). Shadowing is not permitted, i.e. a nested statement block may not declare a variable that has the same name

as a variable that was declared earlier in the parent statement block, or one that has the same name as a global variable.

The variable's name can be any *legal identifier*. A legal identifier starts with an uppercase letter, or a lowercase letter, or a number sign ("#"), and consists of uppercase letters, lowercase letters, digits, and underscores. QQL keywords are not permitted as identifiers. By convention, only global variable identifiers have a hash sign as their first character.

A QQL variable is declared with a type identifier that determines what kind of value the variable can store. QQL defines four primitive variable types: boolean, string, number, and date. The corresponding QQL type identifiers are Bool, String, Numeric, and Date. A QQL variable can also be a reference to a quarc repository entity: quarc, attribute type, attribute enumeration item, quarc relation, communication relation, or communication segment. The corresponding QQL type identifiers are Quarc, Attribute, AttrEnumItem, QuarcRelation, CommRelation, and Segment. Finally, QQL defines a list type with the type identifier List. A QQL list can only contain entities of one single type which is determined when the list is created (see subsection 3.2.1).

```
variabledeclaration ::= vartype variable ( '=' expression )?
vartype ::= 'Bool' | 'String' | 'Numeric' | 'Date' | 'Quarc' | '
   Attribute' | 'AttrEnumItem' | 'QuarcRelation' | 'CommRelation' | '
   Segment' | 'List'
variable ::= identifier
```

Listing 3.5: QQL variables

3.1.5 Conditional Branches

QQL supports a standard if statement. It avoids the "dangling else" problem by requiring statement blocks to always start and end with curly braces, so it is clear for each statement to which block it belongs.

```
if ::= 'if' '(' expression ')' statementblock ( 'else' statementblock
    )?
```

Listing 3.6: QQL conditional branches

3.1.6 Loops

QQL supports standard *while* loops and a list enumeration command, for.

```
while ::= 'while' '(' expression ')' statementblock
for ::= 'for' vartype? variable 'in' expression statementblock
```

Listing 3.7: QQL loops

The while statement first evaluates its expression. If it evaluates to false, the statement block is skipped. If it evaluates to true, the statement block gets executed, afterwards the expression

is evaluated again. This will be repeated until the expression evaluates to false.

The for statement executes its statement block for each element in its expression, which must evaluate to a list. In each iteration, the current list element gets assigned to variable prior to execution of the statement block. The variable can be optionally declared within the for statement. In this case, it is only visible in the statement block. As a consequence, the same variable name can be declared in subsequent for loops in the same parent statement block without shadowing the previous declaration. In all cases, the type of *variable* must match the type of the list elements.

3.1.7 Functions

Functions in QQL consist of a *function header* and a statement block. The function header declares a return type, a function name, and zero or more parameters. Parameters are passed by value, not by reference. They can be used like variables in the function statement block. Parameters are typed like variables, but unlike variables, they can be declared as type Entity which means that any type is acceptable for this parameter. If a return type is declared, the statement block must end with a return statement.

```
functiondefinition ::= 'function' vartype? functionname '(' (
   parameter ( ',' parameter )* )? ] ')' statementblock
functionname ::= identifier
parameter ::= ( 'Entity' | vartype ) identifier
return ::= 'return' [ expression ]
functioncall ::= functionname '(' ( expression ( ',' expression )* )?
    ')'
```

Listing 3.8: QQL functions

3.2 System Functions

QQL defines a number of system functions for operations on lists and management of quarcs, attribute types, attribute values, quarc relations, and communication relations. Many of these functions are required to enable dynamic tailoring. For each function, we show its definition and a brief description.

3.2.1 List Management

Lists are an important concept in QQL. Quarc relations and communication relations are retrieved from quarcs as lists, for example. The elements in a QQL list must all be the same type, which is determined when the list is created. Lists can contain all QQL variable types. We define the following list query and manipulation commands:

function List createList*Type*()
This group of functions create empty Lists for elements of type *Type*.

function List duplicateList(List l)
Creates a duplicate of an existing list.

function removeFromList(List l, Entity x)
Removes entity x from list l. Throws a runtime error if l does not contain x.

function appendToList(List l, Entity x)
Append entity x to list l. Throws a runtime error if the type of x doesn't match the element type of l.

function appendToList(List l, List l2)
Append list l2 to list l. Throws a runtime error if the element type of l2 doesn't match the element type of l.

function Bool contains(List l, Entity x)
Checks if list l contains entity x. Throws a runtime error if the type of x doesn't match the element type of l.

function Numeric listSize(List l)
Returns the number of entities in list l.

The following example demonstrates the use of the QQL list functions. Its goal is to create a list of all quarcs to which quarc q1 and quarc q2 both have a startDependsOn relation. It first creates a new empty list for Quarc entities named start_shared. Next, it obtains the list of all quarcs to which q1 has a startDependsOn relation and stores it in start_q1. It repeats that step for quarc q2 and stores the result in start_q2. Finally, it walks through start_q1, stores each element that is also member of start_q2 in sharedStart, and removes it from start_q1 and start_q2. At the end, start_shared contains all quarcs to which both q1 and q2 have a startDependsOn relation. start_q1 contains the quarcs to which q1 has a startDependsOn relation, but not q2. start_q2 contains the quarcs to which q2 has a startDependsOn relation, but not q1.

```
List start_shared = createListQuarc();
List start_q1 = quarcsRelatedFromQuarc(q1, #startDependsOn);
List start_q2 = quarcsRelatedFromQuarc(q2, #startDependsOn);
for Quarc x in start_q1 {
  if (contains(start_q2, x)) {
    appendToList(start_shared, x);
    removeFromList(start_q1, x);
    removeFromList(start_q2, x);
  }
}
```

Listing 3.9: QQL list functions example

3.2.2 Quarc Management

The following functions can be used to list, create and remove quarcs.

function List allQuarcs()
List of all quarcs in the quarc repository.

function Quarc createQuarc()
Creates a new, empty quarc.

function removeQuarc(Quarc q)
Deletes quarc q. A runtime error is thrown if any relations exist from other quarcs to q. Relations from q to other quarcs and communication relations are deleted automatically.

function removeAndDisconnectQuarc(Quarc q)
Deletes quarc q after removing all existing relations from other quarcs to q.

3.2.3 Attribute Types

The following functions manage attribute type definitions. Attribute definitions are identified and retrieved by a tag in string form. The purpose of these functions is to support dynamic tailoring, by allowing the modification of attribute types in a quarc repository at runtime.

The first three functions create new numeric, date and enumerated attribute types, respectively.

function Attribute createAttributeNumeric(String t)
Creates a new numeric attribute and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned.

function Attribute createAttributeDate(String t)
Creates a new date attribute and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned.

function Attribute createAttributeEnum(String t)
Creates a new enumerated-type attribute and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned.

The next four functions handle enumeration items for enumerated attribute types.

function AttrEnumItem createAttributeItem(String t, Attribute a)
Creates a new enumeration item for attribute a and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned, or if a is not an enumerated attribute type.

function AttrEnumItem findAttributeItemByTag(String t, Attribute a)
Finds the enumeration item type that is tagged with t on attribute a. Throws a runtime error if there is no matching enumeration item type.

function List attributeItems(Attribute a)
Returns an AttrEnumItem-type List of all enumeration items for attribute a. Throws a runtime error if a is not an enumerated attribute type.

function Bool isInOrder(Attribute a, AttrEnumItem v1, AttrEnumItem v2)
Checks if enumeration item v1 is lower in order than v2. Enumeration item order is determined by the order in which they were created on attribute a. Throws a runtime error if v1 or v2 are not enumeration items of attribute a.

The final two functions retrieve an attribute type by its tag, and remove an attribute type entirely.

function Attribute findAttributeByTag(String t)
Finds the attribute type that is tagged with t.

function removeAttribute(Attribute a)
Removes attribute type a. Throws a runtime error if values for a are assigned to any quarcs.

The following example demonstrates the use of the QQL attribute type functions. Its goal is to create a new numeric attribute type DistanceToTarget and for each quarc q, assign to it the distance from q to the quarc target on the quarc graph for the relation startDependsOn. The code creates the attribute, then starts walking the startDependsOn relation at the target quarc, target, which has a distance of 0 to itself. The function setQuarcDistance function sets the distance attribute on the current graph, then retrieves all quarcs to which the current quarc has a startDependsOn relation, and calls itself recursively on each of them with a distance increased by 1. The recursion terminates when there are no startDependsOn relations going out from a quarc.

```
function setQuarcDistance(Quarc q, Numeric distance) {
  Attribute dist_attr = findAttributeByTag("DistanceToTarget");
  assignAttributeValueToQuarc(q, dist_attr, distance);

  for (Quarc x in quarcsRelatedFromQuarc(q, #startDependsOn)) {
    setQuarcDistance(x, distance + 1);
  }
}

function createDistance() {
  Attribute dist = createAttributeNumeric("DistanceToTarget");
  setQuarcDistance(#target, 0);
```

```
}
```

Listing 3.10: QQL attribute type functions example

3.2.4 Attribute Values

The following functions assign and query attribute values on quarcs.

function assignAttributeValueToQuarc(Quarc q, Attribute a, AttrEnumItem v)
Assigns value v for enumerated-type attribute a to quarc q.

function assignAttributeValueToQuarc(Quarc q, Attribute a, Numeric n)
Assigns numeric value n for numeric attribute a to quarc q.

function assignAttributeValueToQuarc(Quarc q, Attribute a, Date d)
Assigns date value d for date attribute a to quarc q.

function Bool isAttributeValueAssignedToQuarc(Quarc q, Attribute a)
Checks if a value is assigned to quarc q for attribute a.

function AttrEnumItem getAttributeValueEnumForQuarc(Quarc q, Attribute a)
Retrieves the value that is assigned to quarc q for enumerated-type attribute a. Returns null if there is no value assigned to q for a.

function Numeric getAttributeValueNumericForQuarc(Quarc q, Attribute a)
Retrieves the value that is assigned to quarc q for numeric attribute a. Returns null if there is no value assigned to q for a.

function Date getAttributeValueDateForQuarc(Quarc q, Attribute a)
Retrieves the value that is assigned to quarc q for date attribute a. Returns null if there is no value assigned to q for a.

function removeAttributeValueFromQuarc(Quarc q, Attribute a)
Remove value for attribute a, if present, from quarc q.

function List quarcsByAttributeValue(Attribute a, AttrEnumItem v)
List of all quarcs in the quarc repository that have value v set for enumerated-type attribute a.

3.2.5 Quarc Relation Types

The following functions manage quarc relation types. Quarc relation definitions are identified and retrieved by a tag in string form. The purpose of these functions is to support dynamic tailoring, by allowing the modification of quarc relation types in a quarc repository at runtime.

function QuarcRelation createQuarcRelation(String t)
Creates a new quarc relation type and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned.

function removeQuarcRelation(QuarcRelation r)
Removes quarc relation type r. Throws a runtime error if any relations between quarcs with this quarc relation type still exist.

function QuarcRelation findQuarcRelationByTag(String t)
Finds the quarc relation that is tagged with t.

3.2.6 Quarc Relations

The following functions manage quarc relations.

function establishQuarcRelation(QuarcRelation r, Quarc q1, Quarc q2)
Establishes quarc relation of type r from quarc q1 to quarc q2.

function disestablishQuarcRelation(QuarcRelation r, Quarc q1, Quarc q2)
Removes quarc relation of type r from quarc q1 to quarc q2.

function List quarcsRelatedToQuarc(Quarc q, QuarcRelation r)
Lists all quarcs x for which a quarc relation of type r exists from x to quarc q.

function List quarcsRelatedFromQuarc(Quarc q, QuarcRelation r)
Lists all quarcs x for which a quarc relation of type r exists from quarc q to x.

function List quarcRelationRoots(r)
Lists all quarcs in the quarc repository that are roots for relation type r, i.e. all quarcs x for which there is no quarc y with a relation of type r from y to x.

function List quarcRelationLeaves(r)
Lists all quarcs in the quarc repository that are leaves for relation type r, i.e. all quarcs x for which there is no quarc y with a relation of type r from x to y.

3.2.7 Communication Relation Types

The following functions manage communication relation types. Communication relation definitions are identified and retrieved by a tag in string form. The purpose of these functions is to support dynamic tailoring, by allowing the modification of communication relation types in a quarc repository at runtime.

function CommRelation createCommRelation(String t)
Creates a new communication relation type and assigns tag t to it. Throws a runtime error if another entity already has tag t assigned.

function removeCommRelation(CommRelation r)
Removes communication relation type r. Throws a runtime error if any relations between quarcs with this communication relation type still exist.

function CommRelation findCommRelationByTag(String t)
Finds the communication relation that is tagged with t.

3.2.8 Communication Relations

The following functions manage communication relations, i.e. relations between quarcs and communication segments. QQL defines no facilities for the management and creation of communication segments and communication artifacts. These capabilities are delegated to concrete implementations of the QUARC metamodel. Communication segments are identified by a string ("csid").

function Segment findCommSegment(String csid)
Finds the communication segment identified by csid.

function establishCommRelation(CommRelation r, Quarc q, Segment s)
Establishes communication relation of type r from quarc q to communication segment s.

function disestablishCommRelation(CommRelation r, Quarc q, Segment s)
Removes communication relation of type r from quarc q to communication segment s.

function List quarcsRelatedToSegment(Segment s, CommRelation r)
Lists all quarcs x for which a communication relation of type r exists from x to communication segment s.

function List segmentsRelatedFromQuarc(Quarc q, CommRelation r)
Lists all communication segments x for which a communication relation of type r exists from quarc q to x.

3.2.9 Validators

Validators are executed in the context of a transaction (see sections 3.1.1 and 3.1.2) to check structural integrity constraints. To do this, they need access to the state of the quarc repository right before the start of the transaction. We define the following system functions, available only in a validator statement block:

function Quarc previous(Quarc q)
Returns a read-only version of quarc q in its state before the start of the transaction.

function List changedQuarcs()
Returns a list of quarcs that changed during execution of the transaction. Changes include attribute value changes and relation changes.

function List removedQuarcs()
Returns a list of quarcs that were removed during the transaction.

function assert(Bool b)
If Boolean expression b evaluates to false, aborts the transaction.

CHAPTER 4

Instantiating the QUARC Metamodel

In this chapter, we demonstrate the power and expressiveness of the QUARC metamodel by showing how to model various processes and tools as QUARC metamodel instances. We cover well known lifecycle processes (Waterfall, Unified Process, and Scrum), process functions (risk management, task management), and tools (bug trackers).

For some of the examples, we illustrate their characteristics by showing a similar software project scenario across all instances. The scenario is the software upgrade from version 8 to version 9 of the mobile dictation software application *Dictamus*, subsequently called the *Dictamus upgrade project*. The goal of the this upgrade project is to provide the following new features.

- **Faster backup** To speed up iTunes backups, a dictation should be stored in several files of 5 MB maximum each, instead of one large file.
- **New audio format** It should be possible to share dictations in another compressed format. Candidates are AAC, MP3, and Speex.
- **Adjustable format** Instead of just switching between compressed and non-compressed sharing, the user needs to be able to select an audio format.
- **Auto delete** The user should have the option to enable a function that automatically deletes dictations when they were shared successfully.
- **Background recording** It should be possible to record audio while the app is in the background.
- **Background sharing** It should be possible to share dictations while the app is in the background.
- **Fast dictation switch ("Instant Save")** There should be no explicit "Save" operations for dictations anymore. Dictations should be saved automatically at all times to allow quick switching between different dictations.

- **Storage Rule Violation Risk** The platform provider will probably soon issue new disk storage rules. The risk is that our chosen implementation violates these rules, and needs to adapted.

4.1 Issue/Rationale Management

4.1.1 IBIS

A "wicked problem" is an ill-defined, complex problem with multiple stakeholders. Rittel & Kunz were the first to tackle wicked problems in the context of design. Their Issue-Based Information System (IBIS) [KR70] describes an issue model and a design method. The model's core entities are *issues* in the form of questions that need to be answered. For each issue, there are *positions* that represent possible solutions or answers. *Arguments* support or object to positions. Arguments can question existing issues, or suggest new ones. Positions can also question and suggest issues. Issues can be generalized or replaced by other issues. Figure 4.1 shows an UML class diagram of IBIS entities and relations.

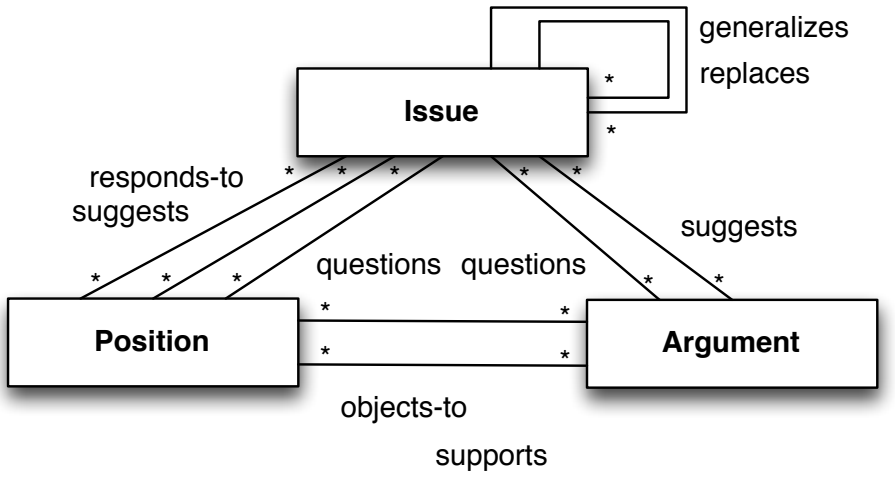


Figure 4.1: IBIS entities and relations as UML class diagram [BD10]
This UML class diagram shows the IBIS entities and relations.

IBIS focuses only on design, while the QUARC metamodel encompasses all the phases of a project. The Quarc entity is a generalization of the Issue entity, because it is not restricted to questions. It can also represent work items. The entire IBIS entity model can be represented in a QUARC repository model, as shown in figure 4.2. In this instantiation, Issue and Position are quarcs. Argument is a communication segment. Issue and Position are both quarcs, so relations between them are quarc relations. Relations between Issue or Position and Argument are communication relations.

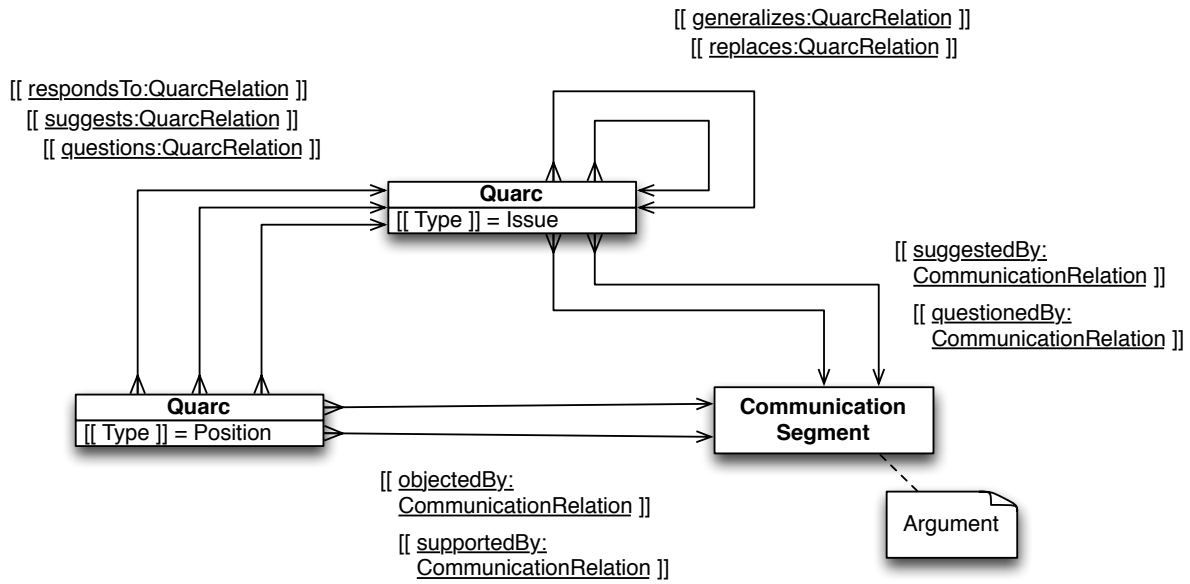


Figure 4.2: Quarc repository model for IBIS

The IBIS entities and relations from fig. 4.1 can be represented in a quarc repository, as shown by this quarc repository model.

Figure 4.3 shows how a subset of the design challenges in the Dictamus upgrade project would be modeled as IBIS entities in a QUARC repository. The main issue is a decision about support for a new compressed audio format. Three solution alternatives are considered, with three arguments that affect each of the candidates. Two of the solution alternatives would bring up one new issue each.

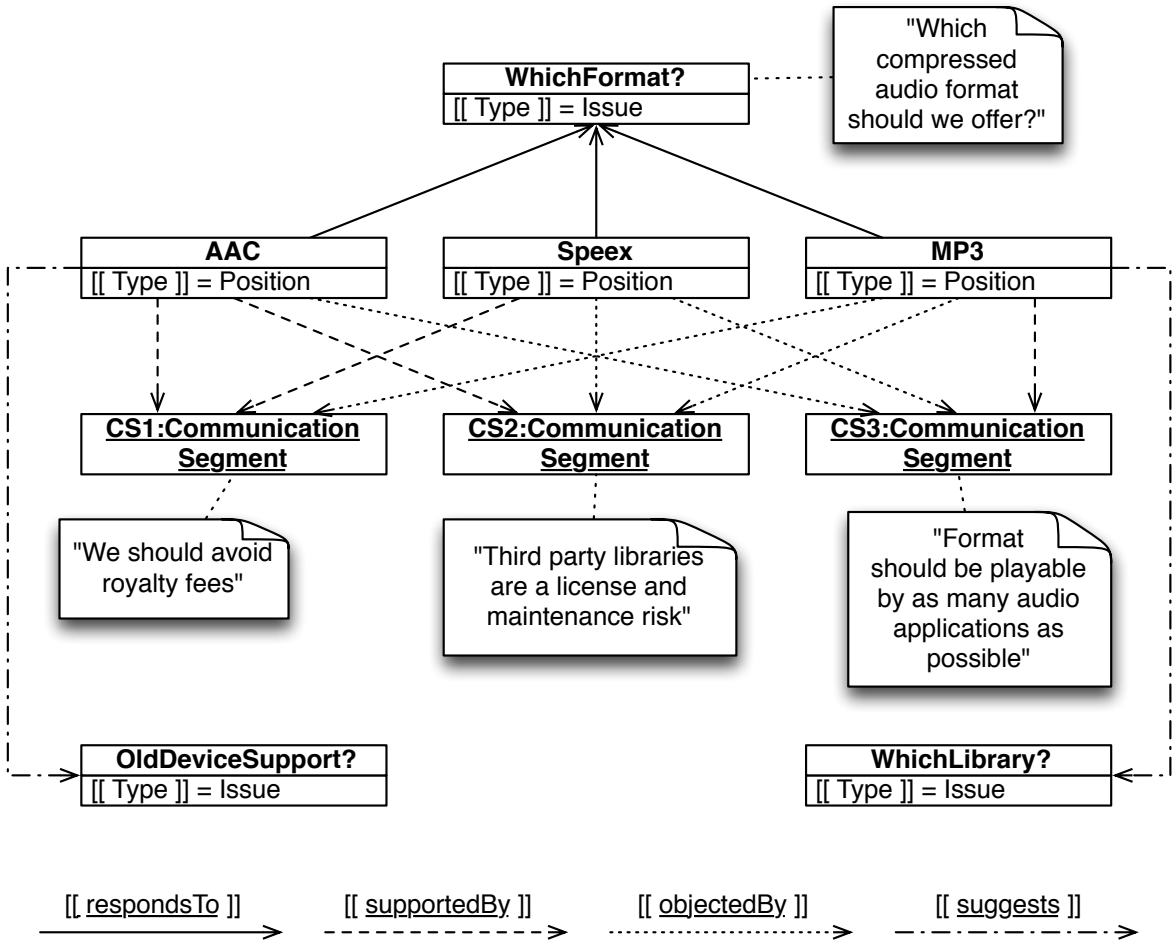


Figure 4.3: Issue from the Dictamus project in IBIS representation

This diagram shows an excerpt from a representation of the Dictamus upgrade project in a quarc repository organized after the model from fig. 4.2. It contains the main issue quarc for this example, three position quarcs as possible solutions, and three argument communication segments. The quarcs and communication segments bear relations that indicate if an arguments supports a position or objects to it. Two positions suggest followup issues.

4.1.2 QOC

Design Space Analysis is an approach to representing design rationale invented by MacLean et al. It shows the design space around an artifact with a notation based on *Questions*, *Options*, and *Criteria* (QOC) ([MYM89], [MYBM91]). Questions identify key design issues, Options provide possible answers to the Questions, and Criteria serve to assess and compare the Options. Arguments discuss the correctness of assessments of Options with respect to Criteria, and Arguments can also discuss Arguments. Figure 4.4 shows a UML model of the entities in QOC. We include the intermediate Assessment class, which is not in the original QOC model, to be able to represent in UML the relationships between Argument and the connection between Option and Criterion (i.e. the Assessment).

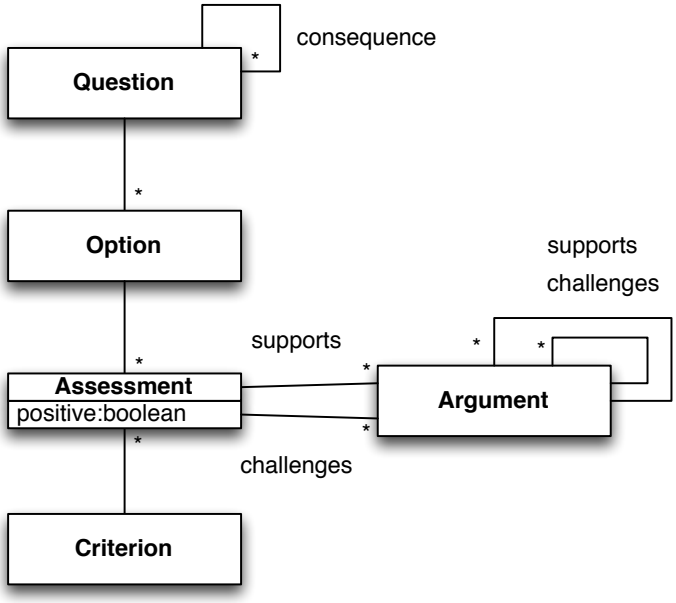


Figure 4.4: QOC entity model

Entities from the Questions, Options, Criteria (QOC) model in a UML class diagram representation. We introduced the Assessment class to be able to model relations between arguments and assessments (relations between options and criteria).

The complex Argument structure is a challenge to transfer to the QUARC metamodel. Representing Questions and Options by quarcs is straightforward, as is representing Criterion and Argument by communication segments. The Assessment of an Option with respect to a Criterion follows as a communication relation, but the QUARC metamodel does not provide facilities to connect communication relations to communication segments. We opt to lose the exact hierarchical dependencies between Arguments themselves and Argument vs. Assessment, and therefore attach the Arguments in an unstructured way to the Option. Figure 4.5 shows the resulting quarc repository model.

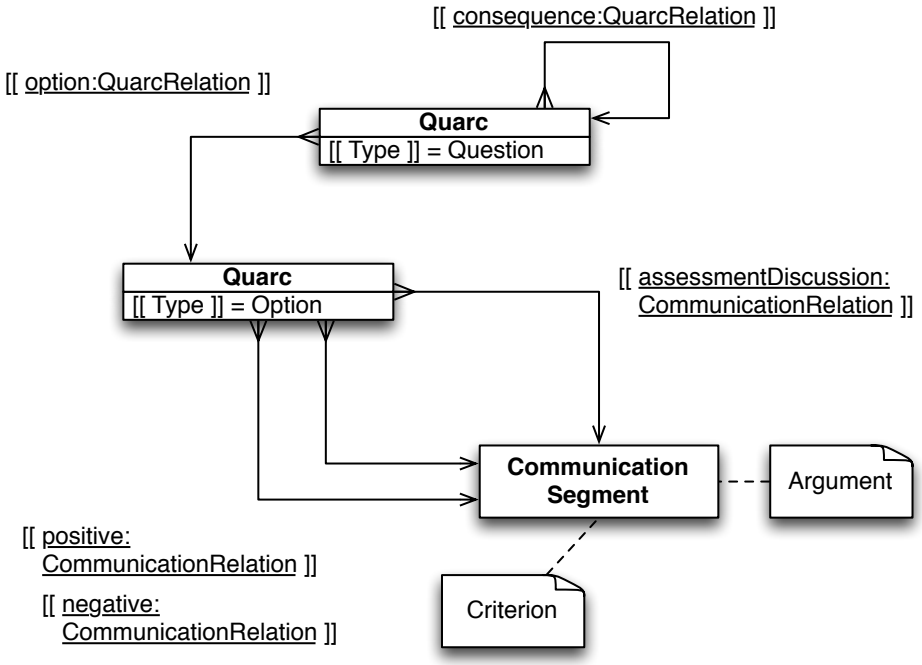


Figure 4.5: Quarc repository model for the QOC entity model

This quarc repository model shows how a QOC model (fig. 4.4) can be represented by quarcs. We reduced the complexity of the Argument/Assessment structure for this representation.

4.1.3 Bug Tracker

A bug tracking system manages a list of bugs and issues in a software project. Bug trackers can be used in all stages of a project, from requirements elicitation to maintenance. They are widely used in projects of all sizes, and there is a wide variety of free and commercial bug tracker software packages available. One of the most popular open source bug trackers is Mantis [Anoa]. Mantis is a web-based bug tracking system written in PHP. The main abstraction in Mantis is the Issue (not to be confused with IBIS issues). Issues can be related to other issues, there are parent/child, duplicate, and unspecified relations between issues. Issues are reported by a User, and they can be assigned to a User. Issues belong to a Project. Projects can have sub-Projects. Figure 4.6 shows the Mantis object model.

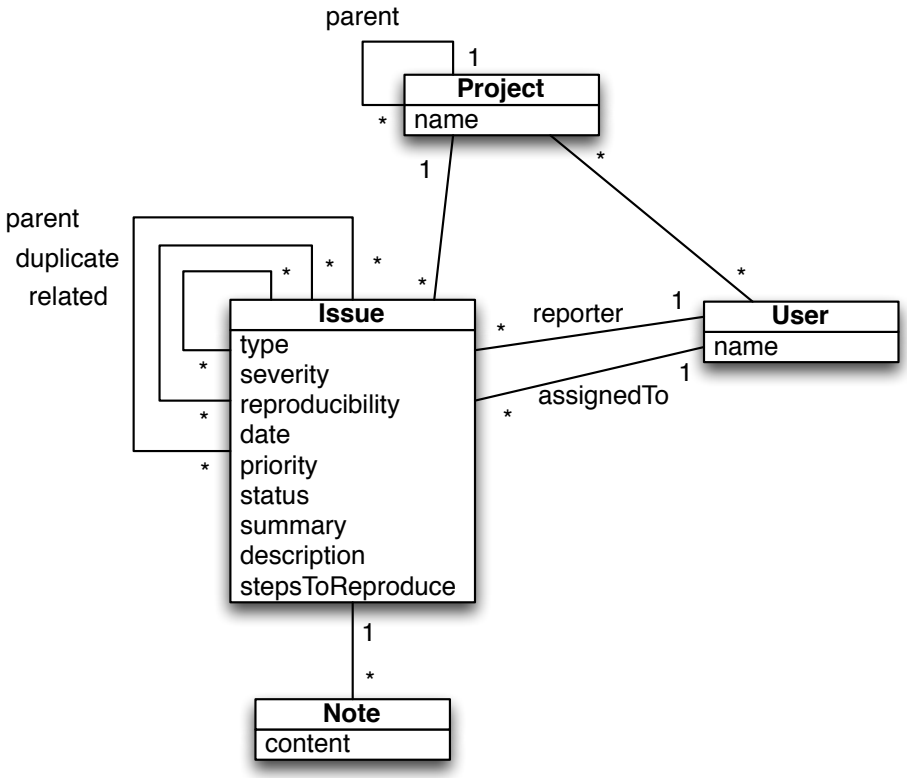


Figure 4.6: Mantis entities and relations

This UML class diagram shows a representation of the entities from the Mantis bug tracking system.

Modeling the Mantis entities as a quarc repository model is straightforward. We represent the Mantis Issue with a quarc. Summary, Description, StepsToReproduce, and Note are text-based information in Mantis, so we model them as communication segments. To connect these communication segments to issues, we define one communication relation per information type. The relations are summary, description, stepsToReproduce, and note. See figure 4.7 for the quarc repository model.

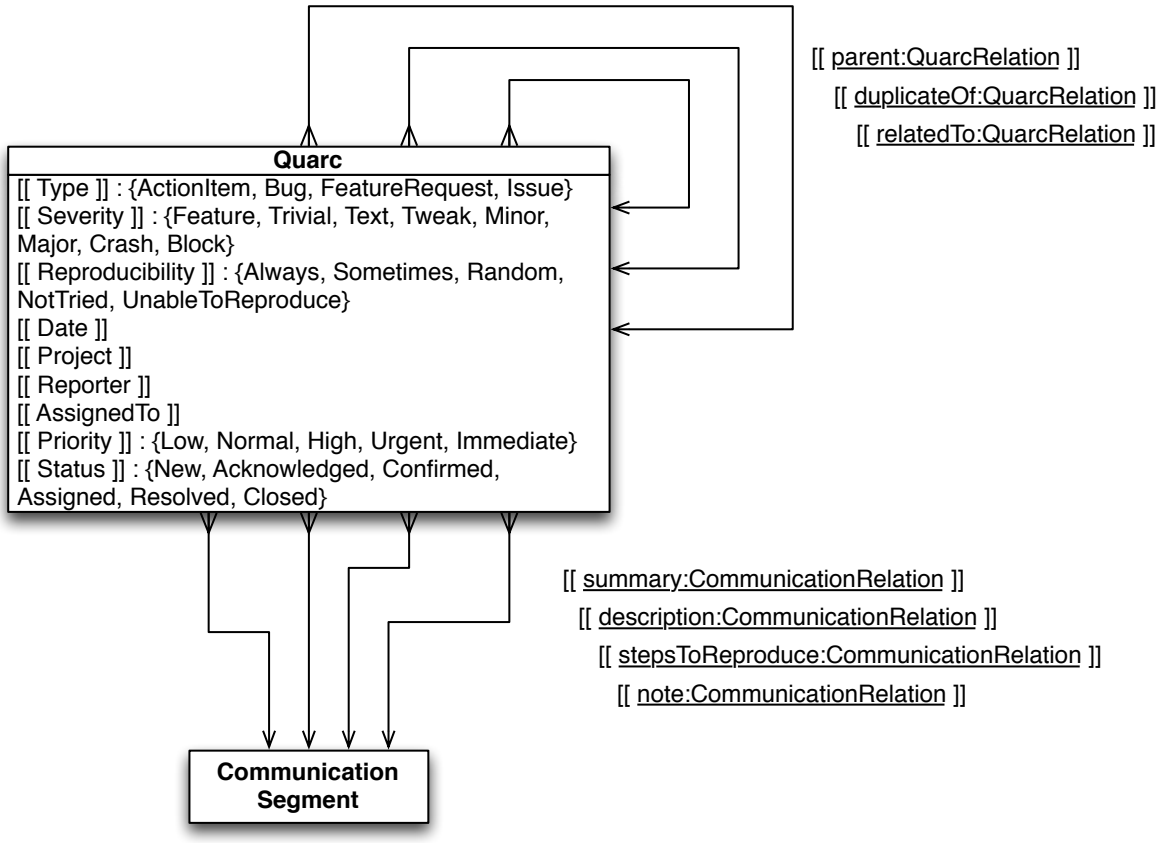


Figure 4.7: Quarc repository model for the Mantis entity model

This quarc repository model shows how the Mantis entities can be represented by quarcs. User and Project are modeled as enumerated attributes.

4.2 Task Management

An important aspect of project management is task management, i.e. identifying tasks in a work breakdown structure, assigning them to project participants, scheduling their execution, and tracking their progress. Relationships between tasks have traditionally been modeled in PERT (Program Evaluation and Review Technique) [MRCF59] charts. PERT and the Critical Path Method (CPM) [KJW59] are frequently used together in project management.

A PERT chart visualizes task dependencies as a directed acyclic graph of tasks. Each task is annotated with the expected time it takes to complete the task, and with planned start and end dates. See fig. 4.8 for an example. From this information, a schedule for the project can be derived.

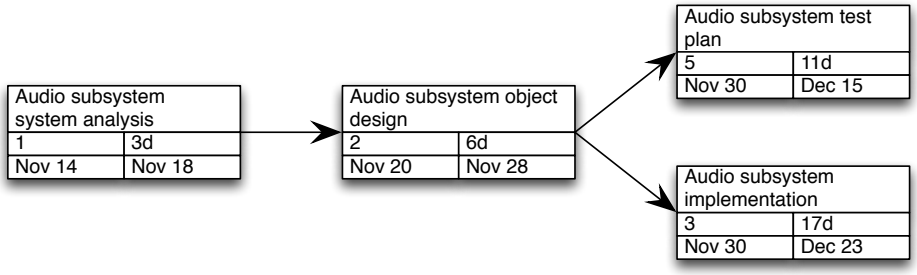


Figure 4.8: Sample PERT chart
Four tasks with estimated durations, start/end dates and dependencies.

The object model of PERT is very simple. It consists of a task that is connected to other tasks. See figure 4.9.

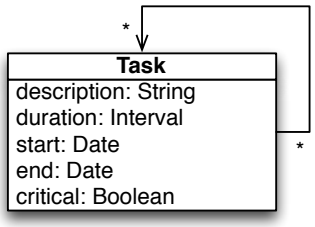


Figure 4.9: PERT entity model
The PERT entities represented as a UML class diagram

In figure 4.10, we show how the PERT entity model can be instantiated from the QUARC metamodel. Naturally, the Task entity is a Quarc instance. Duration, startDate and endDate are already present as default attributes of a quarc (see subsection 2.3.2). The description is a CommunicationSegment instance, with a CommunicationRelation instance as the link between Task and Description.

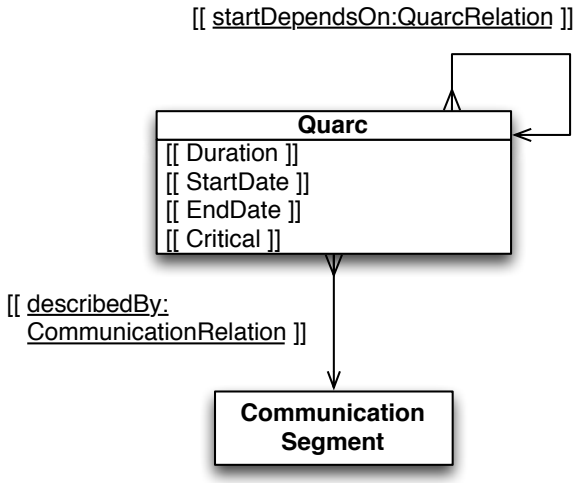


Figure 4.10: Quarc repository model for PERT/CPM
This quarc repository model shows how the PERT/CPM entity (fig. 4.9) can be represented by quarcs.

Figure 4.11 shows how tasks from the Dictamus upgrade project would be modeled in PERT/CPM. This project plan assumes a short runtime and a workforce that can handle at least five tasks in parallel. The diagram shows dates as attribute values for easier readability.

We demonstrate how a quarc repository that represents a PERT/CPM project can update the critical path automatically after each change with a QQL validator. There are three attributes defined in the repository: Duration, which represents a unit-less metric for the amount of effort for each quarc, CriticalPath, which is set to Yes on all quarcs on the critical path, and LSPLength (Longest SubPath Length), which is set on each quarc and indicates the length of longest sub path that starts on this quarc. The paths are determined by the startDependsOn quarc relation.

The validator in listing 4.1 processes the entire repository after each change, determines the current critical paths based on the Duration values and startDependsOn quarc relations, and sets the CriticalPath attribute on the quarcs on the critical paths. There can be several paths with the same length that are longer than all other paths, hence the plural in "critical paths".

The validator starts by identifying *path roots*, i.e. active quarcs (open or in progress) that have no outgoing startDependsOn quarc relations to other active quarcs, because critical paths can only start in a path root. At the same time, it clears the previously assigned values for CriticalPath and LSPLength because they may be invalid after the repository change.

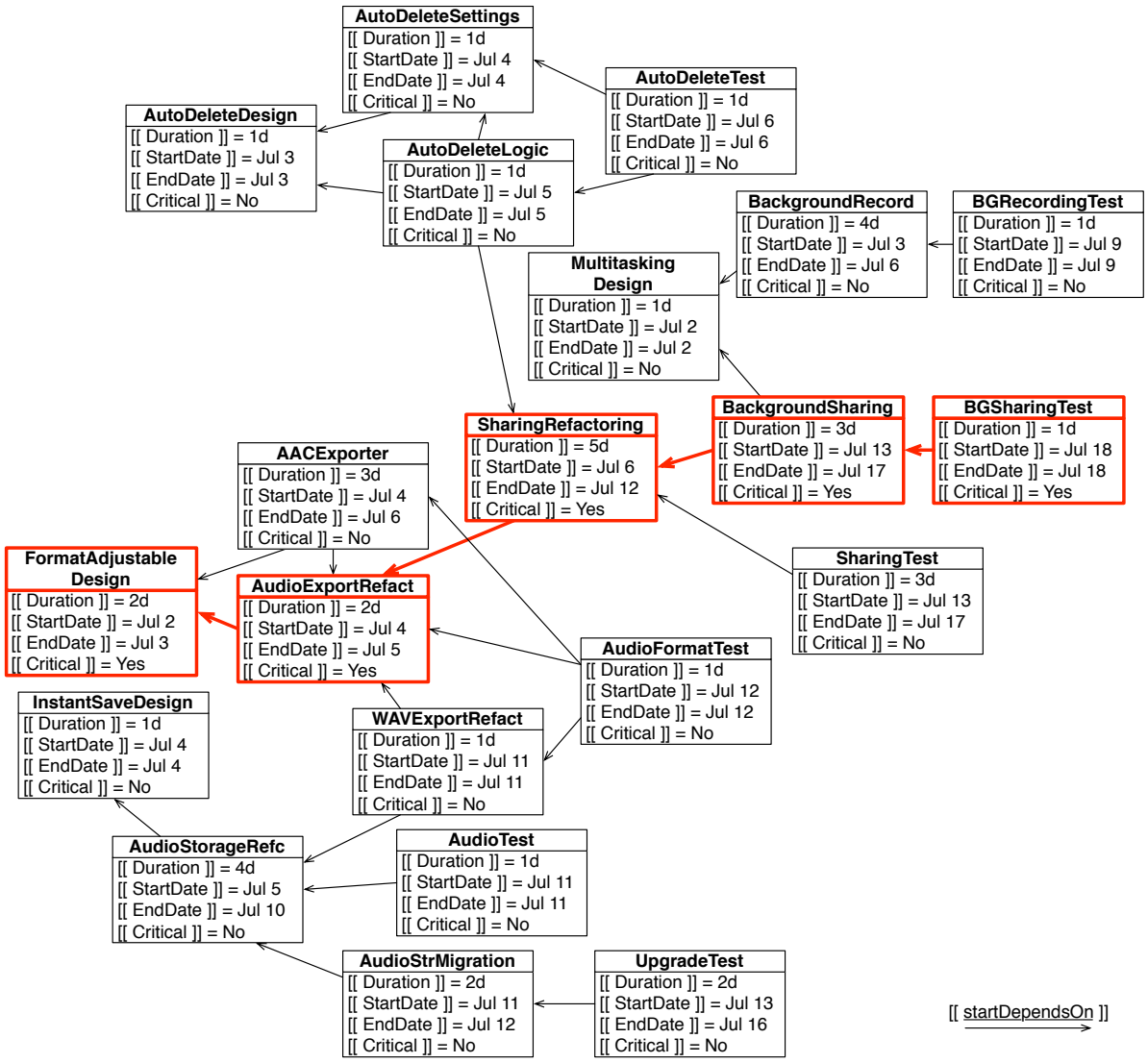


Figure 4.11: Dictamus tasks as PERT/CPM quarcs

This is an excerpt from the Dictamus upgrade project modeled as a PERT/CPM quarc repository. The critical path is shown in red lines (thicker lines in black & white).

Next, the validator descends recursively down all possible paths that start at the path roots, calculates the maximum length of the sub paths for each quarc on its way back from each path end, and stores it in the LSPLength of each quarc. After this step, each path root has a LSPLength value assigned that indicates the length of the longest path starting at that path root. Therefore, the path roots with the biggest length are the path roots of the critical paths.

Finally, the validator walks down each critical path and assigns value Yes for the CriticalPath attribute to all quarcs on the path. In each quarc on the path, the critical sub paths are identified as those paths that have a maximum sub path length that equals the maximum sub path length of the current quarc minus the duration of the current quarc.

```
validator calculateCriticalPath {
  Attribute criticalPath;
  Attribute lspLength;
  criticalPath = findAttributeByTag("CriticalPath");
  lspLength = findAttributeByTag("LSPLength");

  // get path roots and clear previous critical path attributes
  List pathRoots;
  pathRoots = createListQuarc();
  for Quarc x in allQuarcs() {
    // clear attributes
    removeAttributeValueFromQuarc(x, criticalPath);
    removeAttributeValueFromQuarc(x, lspLength);

    // path root?
    AttrEnumItem stat;
    stat = getAttributeValueEnumForQuarc(x, #status);
    if ((stat != #closed) && (stat != #dormant)) {
      List outgoingStartDepends;
      outgoingStartDepends = quarcsRelatedFromQuarc(x, #startDependsOn
          );
      Bool found;
      found = false;
      for Quarc y in outgoingStartDepends {
        stat = getAttributeValueEnumForQuarc(y, #status);
        if ((stat != #closed) && (stat != #dormant)) {
          found = true;
        }
      }
      if (!found) {
        appendToList(pathRoots, x);
      }
    }
  }

  // find biggest path length
  Numeric longestPath;
```

```
  longestPath = 0;
  for (Quarc x in pathRoots) {
    Numeric pathLength;
    calculateLSPLength(x);
    pathLength = getAttributeValueNumericForQuarc(x, lspLength);
    if (pathLength > longestPath) {
      longestPath = pathLength;
    }
  }

  // annotate critical path(s)
  for (Quarc x in pathRoots) {
    Numeric pathLength;
    pathLength = getAttributeValueNumericForQuarc(x, lspLength);
    if (pathLength == longestPath) {
      annotateCriticalPath(x);
    }
  }
}

function void calculateLSPLength(Quarc q) {
  Attribute lspLength;
  lspLength = findAttributeByTag("LSPLength");

  Numeric longestSubPath;
  longestSubPath = 0;

  List dependents;
  dependents = quarcsRelatedToQuarc(q, #startDependsOn);
  for (Quarc x in dependents) {
    AttrEnumItem stat;
    stat = getAttributeValueEnumForQuarc(x, #status);
    if ((stat != #closed) && (stat != #dormant)) {
      Numeric subPathLength;
      if (!isAttributeValueAssignedToQuarc(x, lspLength)) {
        calculateLSPLength(x);
      }
      subPathLength = getAttributeValueNumericForQuarc(x, lspLength);
      if (subPathLength > longestSubPath) {
        longestSubPath = subPathLength;
      }
    }
  }

  Attribute duration;
  duration = findAttributeByTag("duration");

  Numeric pathLength;
```

```
  pathLength = longestSubPath + getAttributeValueNumericForQuarc(q,
     duration);

  assignAttributeValueToQuarc(q, lspLength, pathLength);
}

function void annotateCriticalPath(Quarc q) {
  Attribute criticalPath;
  criticalPath = findAttributeByTag("CriticalPath");
  AttrEnumItem cpYes;
  cpYes = findAttrEnumItemByTag("Yes", criticalPath);

  assignAttributeValueToQuarc(x, criticalPath, cpYes);

  Attribute lspLength;
  lspLength = findAttributeByTag("LSPLength");

  pathLength = getAttributeValueNumericForQuarc(x, lspLength);

  Attribute effort;
  effort = findAttributeByTag("Effort");

  Numeric critSubPathLength;
  critSubPathLength = pathLength - getAttributeValueNumericForQuarc(x
     , effort);

  List dependents;
  dependents = quarcsRelatedToQuarc(q, #startDependsOn);
  for (Quarc x in dependents) {
    AttrEnumItem stat;
    stat = getAttributeValueEnumForQuarc(x, #status);
    if ((stat != #closed) && (stat != #dormant)) {
      Numeric subPathLength;
      subPathLength = getAttributeValueNumericForQuarc(x, lspLength);
      if (subPathLength == critSubPathLength) {
        annotateCriticalPath(x);
      }
    }
  }
}
```

Listing 4.1: QQL validator that calculates the current critical path after each change on the repository

4.3 Risk Management

Risk management consists of risk assessment (identification, analysis, evaluation) and risk treatment (countermeasures, monitoring, mitigation). There are several established risk management processes, such as the approach initially proposed by Barry Boehm [Boe89] [Boe91] or the recent ISO 31000 standard [Int09].

Risk identification is an activity that aims to find as many risks for a project as possible. The identified risks are analyzed to gain a broader understanding of the nature of the risk, its possible consequences, the factors that may trigger the risk, and their probabilities. Next, the consequences and triggers are evaluated to prioritize the risks and to decide on a plan for countermeasures, monitoring and mitigation strategies. Subsequently, the countermeasures get implemented, identified risks are being monitored, and if a risk materializes, mitigation strategies are applied.

The QUARC metamodel supports risk management from two different points of view. The first one is the activities of the risk management process itself. These are tasks that can be represented by quarcs. An umbrella "risk identification" task quarc, for example, may depend on a number of sub-tasks for each subsystem or project activity, each of which would also be represented by a quarc. Risk countermeasures are also tasks that are performed in a project to reduce the likelihood of risks or their consequences. All these tasks can be modeled as quarcs like other project tasks.

The second point of view considers the risk mitigation measures devised in the risk management process. Risk mitigation measures are tasks that will only be performed if a risk actually materializes, in order to deal with its consequences. A risk mitigation measure, should it go into effect, may affect other tasks and decisions in the project, including those that are already closed. Risk mitigation measures are modeled as quarcs in the Dormant state with dependency relations to other quarcs. A dormant quarc is treated like a closed quarc from a dependency enforcement perspective. When a mitigation measure goes into effect because the related risk materializes, its quarc is opened, i.e. its status changes from Dormant to Open or In Progress. If a mitigation measure has an effect on another quarc, the usual way to model this is to establish a closeDependsOn or startDependsOn quarc relation from the affected quarc to the mitigation measure quarc. This means that the affected quarc can only go to status Closed if the mitigation quarc is Closed or Dormant. If an affected quarc is already closed when a risk mitigation quarc needs to be opened, then a situation arises that violates structural integrity constraints. To resolve this, all closed affected quarcs must be reopened. Reopening a closed affected quarc may in turn violate structural integrity constraints due to dependencies on that quarc, so the reopening process must cascade until a state is reached where there are no more structural integrity violations.

In the following set of figures, we show how a hybrid risk/task model behaves when a risk materializes. We omitted the issue management aspects and task management details like start/end date and duration for brevity, as they do not enrich this particular example further. Figure 4.12 shows a snapshot in time of a subset of the Dictamus upgrade project quarcs. Most project quarcs are already closed and only a few are still in progress, which indicates that the project should be completed in the near future. In the top left corner, there is a risk, StorageRisk, which represents the risk that the file storage implementation in the project violates new platform guidelines that are announced, but not yet available. The quarc ReengineerStorage represents the risk mitigation activity that is assigned to this risk.

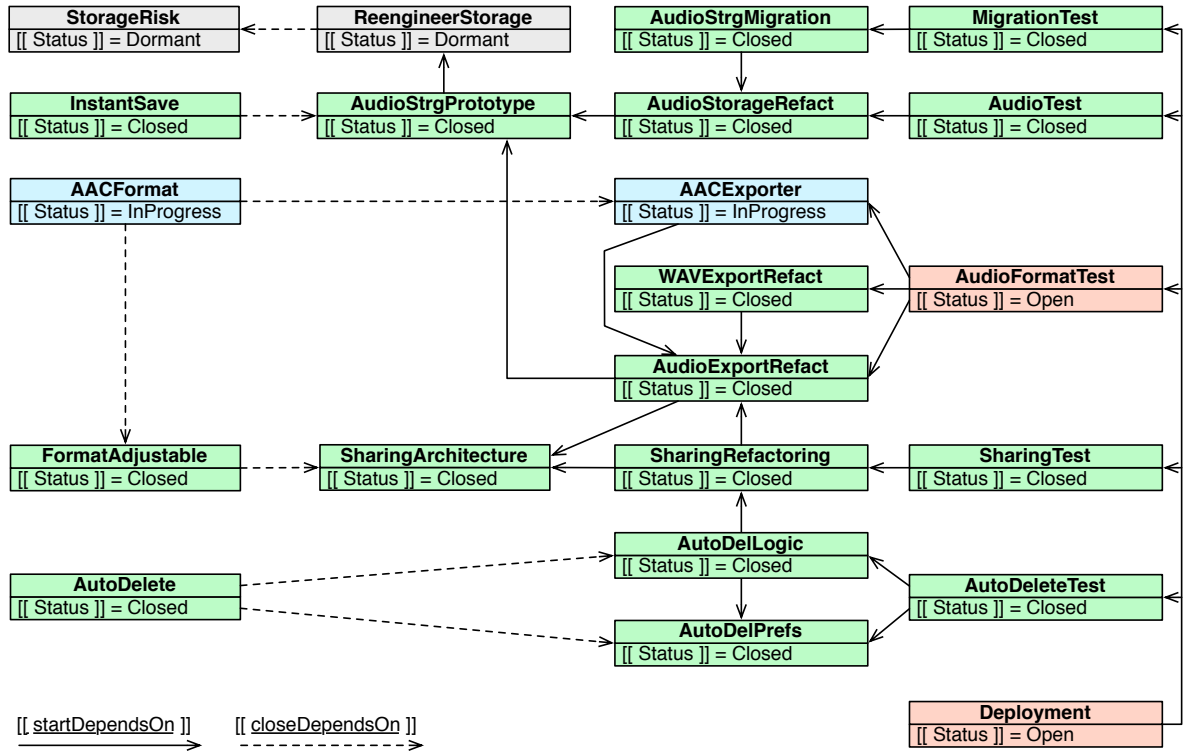


Figure 4.12: Dictamus quarc repository with dormant risk

This diagram shows an excerpt from the Dictamus upgrade project modeled in a default quarc repository (see subsection 2.6.3). The dormant risk StorageRisk expresses that there's a risk that the currently chosen storage model clashes with not-yet-available official guidelines. It has a mitigation measure quarc, ReengineerStorage, which is also dormant.

Now the identified risk materializes. Figure 4.13 shows the quarc repository after the risk StorageRisk and its mitigation measure ReengineerStorage changed their status from Dormant to Open. There is an integrity violation in the quarc repository due to the startDependsOn quarc relation from AudioStrgPrototype to ReengineerStorage. AudioStrgPrototype must be Open or Dormant as long as ReengineerStorage is not Closed or Dormant.

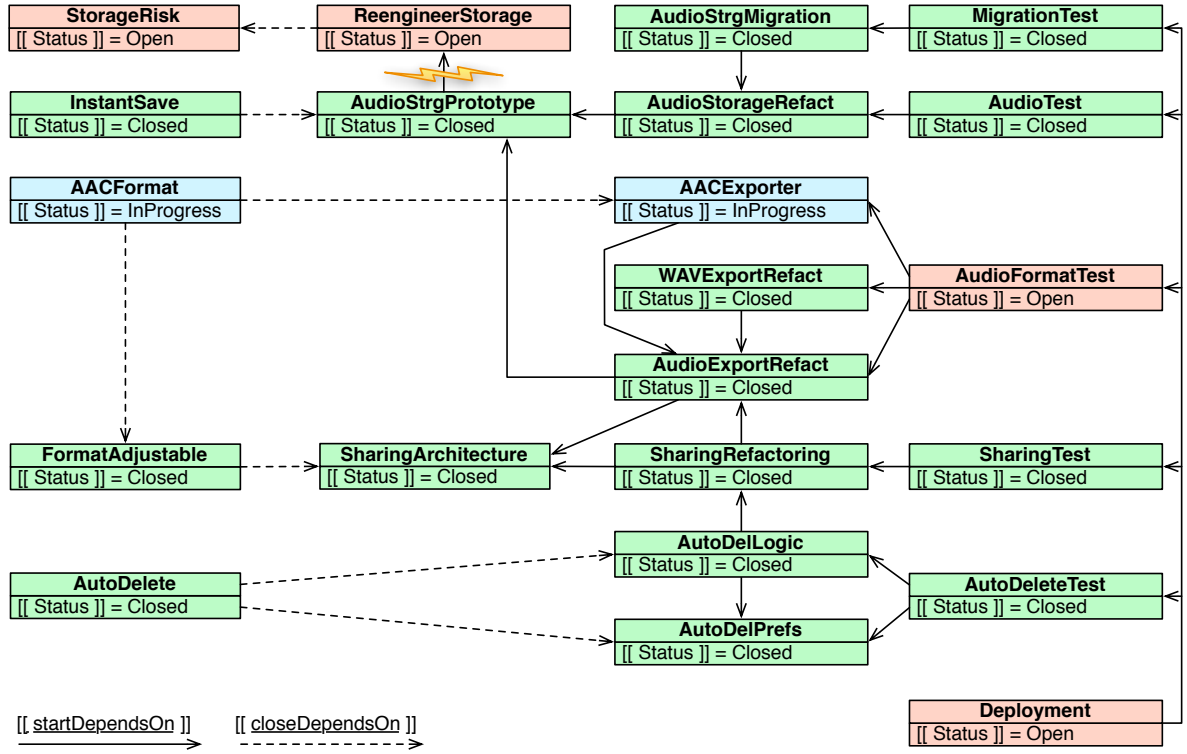


Figure 4.13: Materialized risk causes integrity violation

The StorageRisk materializes, and the ReengineerStorage mitigation measure is force-opened. This causes a structural integrity constraint violation, symbolized by a lightning icon.

In order to resolve the integrity violation, quarc AudioStrgPrototype is also changed to Open. This introduces new integrity violations due to the closeDependsOn quarc relation from InstantSave and the startDependsOn quarc relations from AudioStorageRefact and AudioExportRefact. Figure 4.14 shows the new integrity violations.

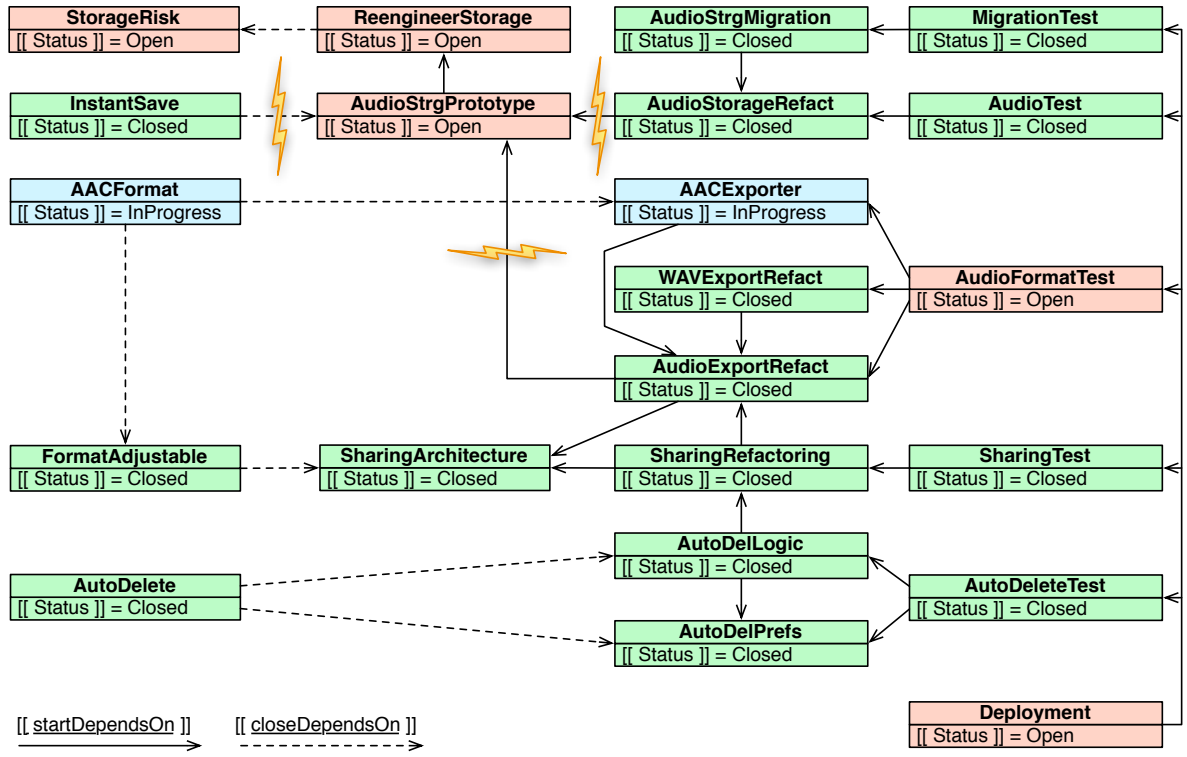


Figure 4.14: Resolving the integrity violation causes new violations

Force-opening the AudioStrgPrototype quarc resolves the structural integrity constraint violation from fig. 4.13, but causes three new violations, also symbolized by lightning icons.

Resolving the three new integrity violations by opening the respective quarcs causes yet other violations. This cascade of integrity violations and resolution continues until the quarc graph finally reaches a consistent state again, as shown in figure 4.15. All of the project quarcs except one are now in status Open again, which indicates that the risk materialization affected almost all tasks in the entire project.

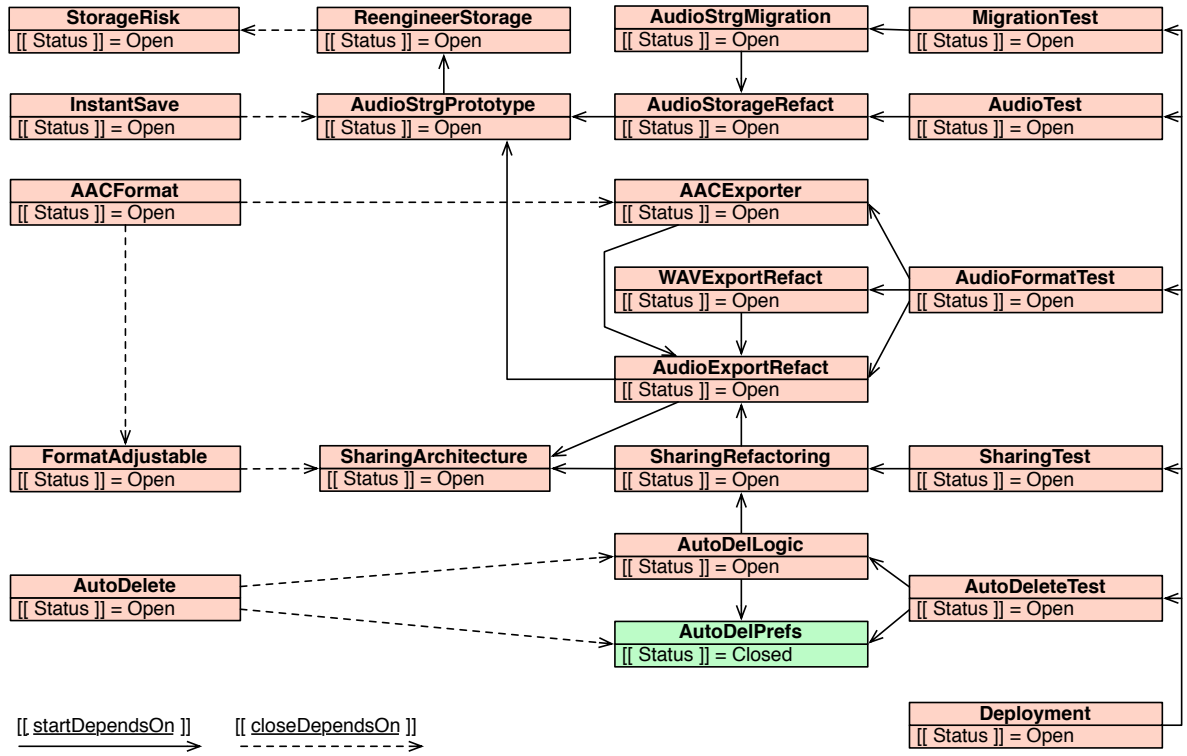


Figure 4.15: All integrity violations resolved

Here, all violations were resolved and all structural integrity constraints are satisfied again. The materialization of the risk caused almost all previously closed project quarcs to revert to status Open.

The default constraints in a quarc repository prevent transactions that result in a state with integrity violations. Therefore, the entire process outlined above must be contained within a single transaction. Listing 4.2 shows a QQL procedure called forceOpen that changes a quarc's status to Open and recursively walks through all incoming closeDependsOn and startDependsOn quarc relations and forces the quarcs on the other end open as well. For closeDependsOn quarc relations, only quarcs with status Closed need to be reopened because InProgress does not violate the relation integrity.

```
function void forceOpen(Quarc x) {
  setAttributeValueEnumForQuarc(x, #status, #open);
  List depC;
  depC = quarcsRelatedToQuarc(x, #closeDependsOn);
  appendToList(dep, depC);
  for Quarc y in depC {
```

```
    AttrEnumItem status_rel;
    status_rel = getAttributeValueEnumForQuarc(y, #status);
    if (status_rel == #closed) {
      forceOpen(y);
  }
  List depS;
  depS = quarcsRelatedToQuarc(x, #startDependsOn);
  appendToList(dep, depS);
  for Quarc y in depS {
    AttrEnumItem status_rel;
    status_rel = getAttributeValueEnumForQuarc(y, #status);
    if ((status_rel != #open) && (status_rel != #dormant)) {
      forceOpen(y);
  }
}
```

Listing 4.2: QQL function that force-opens a risk quarc and reopens related quarcs recursively as necessary to satisfy dependencies

4.4 Software Lifecycle Models

There are several established and widely-used models for the software development process called software life cycle models. The most commonly used are Waterfall Model, V-Model, Spiral Model, Unified Software Development Process and Scrum. The defining characteristics of these life cycle models can be modeled in our quarc-based framework by a combination of specialized tags and constraints.

4.4.1 Linear Models

The waterfall model [Roy70] defines a number of activities that follow each other sequentially (see figure 4.16). V-Model [JT79] is basically an enriched waterfall model with the additional requirement that analysis and design activities produce specifications for associated testing and maintenance activities (see figure 4.17). From the internal perspective of a project, Waterfall and V-Model are equivalent. The main difference lies in the structure of the documents produced in each of the activities.

Figure 4.18 shows a quarc repository that represents the mobile dictation software upgrade project in a waterfall life cycle in the middle of the Verification phase. All quarcs assigned to the Requirements, Design, and Implementation phases are closed, while some of the Verification quarcs are still in progress. The Project quarc represents the current state of the project, it indicates that the current project phase is Verification.

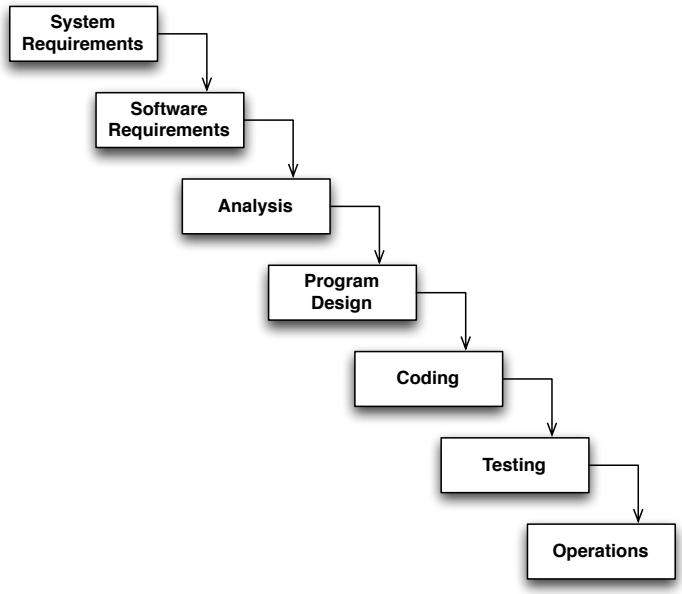


Figure 4.16: Activities in the waterfall model [Roy70]

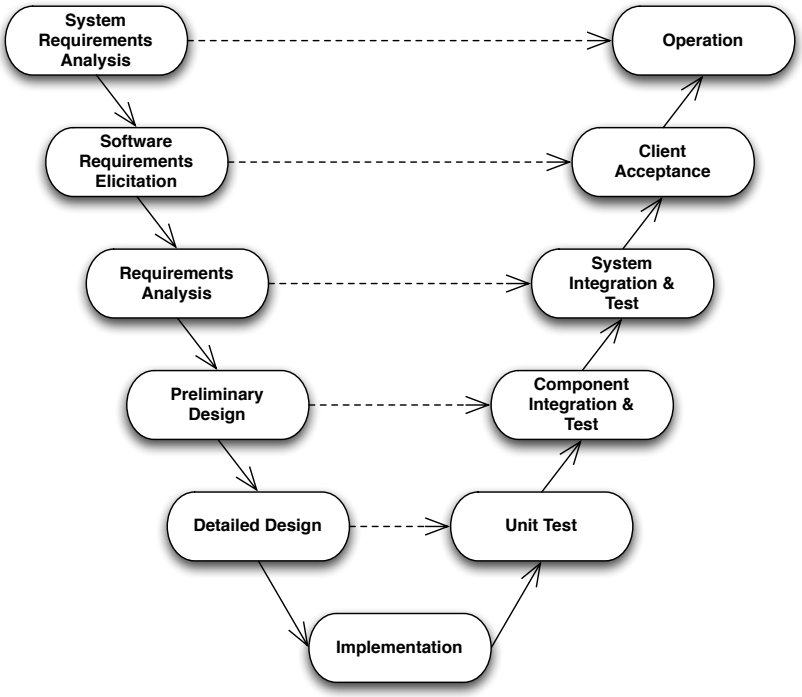


Figure 4.17: Activities in the V-Model [BD10]

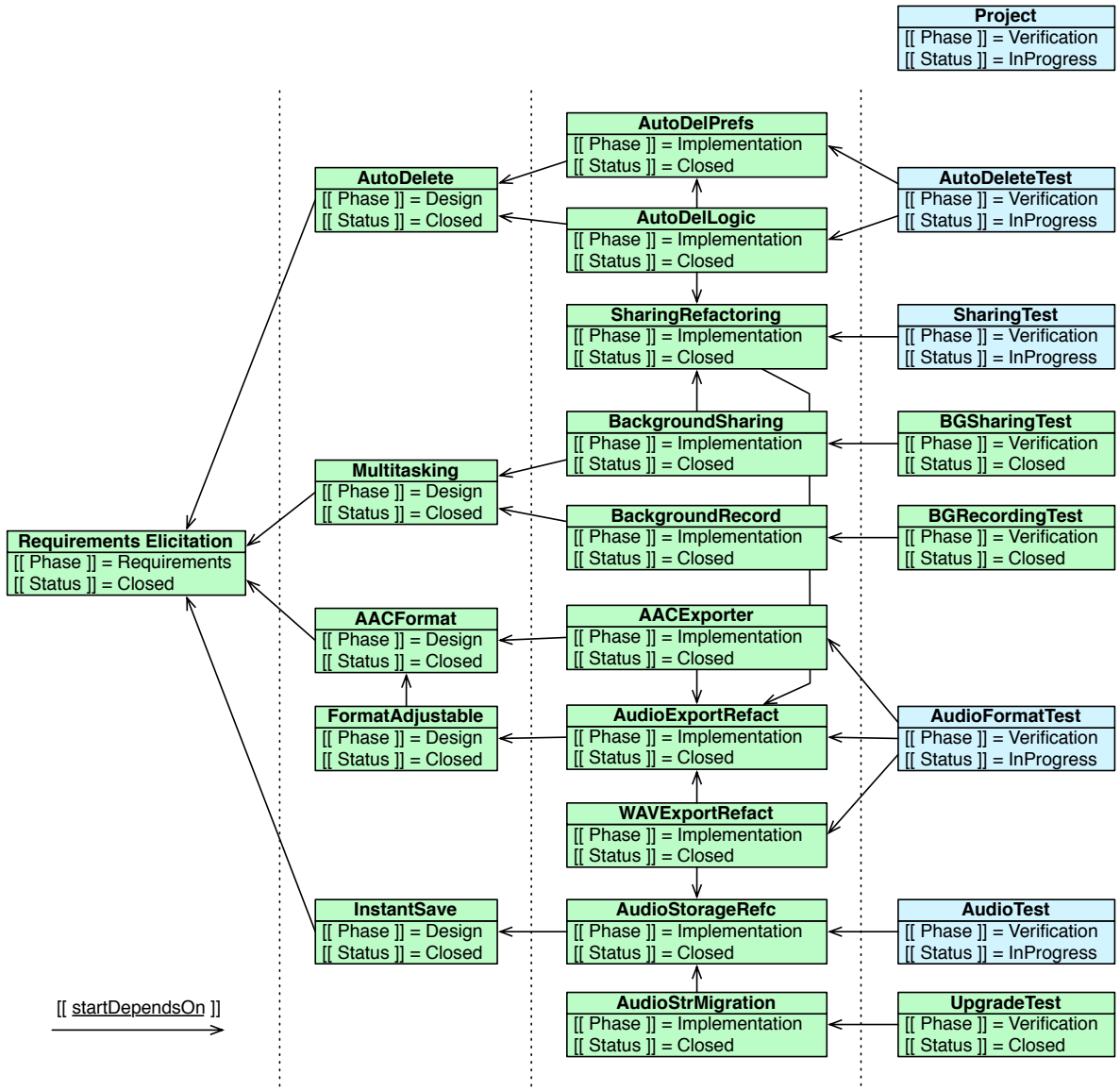


Figure 4.18: Dictamus upgrade project as Waterfall quarcs

This quarc repository represents the Dictamus upgrade project modeled as a waterfall project. The project is currently in the Verification phase, with some verification quarcs already closed and others still in progress. All quarcs assigned to previous phases are closed.

We can enforce waterfall properties in a quarc repository by applying the following predefined quarcs, attributes, and validators to the repository.

The master quarc Project represents the entire project. In order to use the master quarc in validators, we define a global variable #project and assign Project to it. Then, we define an attribute Phase that represents a project activity in the quarc graph timeline. Phase is an enumerated attribute that defines one value for each development activity, e.g. {System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing, Operations} according to fig. 4.16. The value for Phase on the master quarc Project determines the current project activity. We also need to enforce the following constraints on the transitions in the quarc repository.

- Closed quarcs cannot be changed anymore. The only exception is reopening a quarc, which means changing its status to another value than Closed or Dormant. If a quarc is reopened, it must be assigned to the current or a later project activity.
- Quarcs that belong to earlier activities cannot be changed anymore.
- Quarcs cannot be moved to past activities.
- A project can only move to the next activity if all quarcs that are assigned to the current activity have either status Closed or Dormant.

See listing 4.3 for a representation of these constraints as QQL code.

```
function Bool isEarlierThan(Quarc x, Quarc y) {
   AttrEnumItem phaseX = getAttributeValueEnumForQuarc(x, #phase);
   AttrEnumItem phaseY = getAttributeValueEnumForQuarc(y, #phase);
   return isInOrder(phaseX, phaseY);
}

validator dontChangeClosed {
  for Quarc x in changedQuarcs() {
   AttrEnumItem previousStatus;
   AttrEnumItem currentStatus;
   previousStatus = getAttributeValueEnumForQuarc(previous(x), #
      status);
   currentStatus = getAttributeValueEnumForQuarc(x, #status);
   assert((previousStatus != #closed) || ((currentStatus != #closed)
      && (!isEarlierThan(x, #project))));
  }
}

validator dontChangeEarlierActivities {
  for x in changedQuarcs() {
   assert(!isEarlierThan(previous(x), previous(#project));
  }
}
```

```
validator dontMoveToEarlierActivities {
  for x in changed() {
    assert(!isEarlierThan(x, #project);
  }
}

validator verifyAllClosed {
  if (contains(changedQuarcs(), #project)) {
    for x in changedQuarcs() {
      AttrEnumItem currentStatus;
      currentStatus = getAttributeValueEnumForQuarc(x, #status);
      assert((currentStatus == #closed) || (currentStatus == #dormant)
          || (!isEarlierThan(x,#project)));
    }
  }
}
```

Listing 4.3: QQL program that enforces linear model characteristics

4.4.2 Iterative Models

4.4.2.1 Spiral Model

Boehm's spiral model [Boe86] is an activity life cycle model that accommodates infrequent change. It is based on the same activities as the waterfall model, but these activities are repeated in several cycles or rounds.

We can enforce spiral model properties in a quarc repository by applying the following predefined quarcs, attributes and validators to the project.

The master quarc Project represents the entire project. In order to use the master quarc in validators, we define a global variable #project and assign Project to it. Then, we define an attribute ProjectRound that represents a project round in the quarc graph timeline. ProjectRound is an enumerated-type attribute that defines one value for each round, such as {Concept of Operation, Software Requirements, Software Product Design, Detailed Design, Code, Unit Test, Integration and Test, Acceptance Test, Implementation} [BD10]. The value for ProjectRound on the master quarc Project determines the current project round. Furthermore, we define an attribute RoundActivity that represents an activity within a round. RoundActivity is an enumerated-type attribute that defines one value for each activity, for example {Determine Objectives, Specify Constraints, Generate Alternatives, Identify Risks, Resolve Risks, Develop and Verify Next-level Product, Plan}. The value for roundActivity on the master quarc Project determines the current round activity. We also need to enforce the following constraints on quarc repository transitions. They are similar to the linear model constraints in subsection 4.4.1.

- Closed quarcs cannot be changed anymore. The only exception is reopening a quarc, which means changing its status to another value than Closed or Dormant. If a quarc is reopened, it must be assigned to a later project round, or to the current round and the current or a later activity.

- Quarcs that belong to earlier rounds or activities cannot be changed anymore.
- Quarcs cannot be moved to earlier rounds or activities.
- A project can only move to the next round or activity if all quarcs that are assigned to the current activity in the current round have either status Closed or Dormant.

See listing 4.4 for a representation of these constraints as QQL code. The only difference to the linear model code in listing 4.3 is the isEarlierThan function.

```
function Bool isEarlierThan(Quarc x, Quarc y) {
   AttrEnumItem roundX = getAttributeValueEnumForQuarc(x, #
      projectRound);
   AttrEnumItem roundY = getAttributeValueEnumForQuarc(y, #
      projectRound);
   if (isInOrder(roundX, roundY)) return TRUE;
   if (isInOrder(roundY, roundX)) return FALSE;
   AttrEnumItem activityX = getAttributeValueEnumForQuarc(x, #
      projectActivity);
   AttrEnumItem activityY = getAttributeValueEnumForQuarc(y, #
      projectActivity);
   return (isInOrder(roundX, roundY) || ((roundX == roundY) &&
      isInOrder(activityX, activityY)));
}

validator dontChangeClosed {
  for Quarc x in changedQuarcs() {
   AttrEnumItem previousStatus;
   AttrEnumItem currentStatus;
   previousStatus = getAttributeValueEnumForQuarc(previous(x), #
      status);
   currentStatus = getAttributeValueEnumForQuarc(x, #status);
   assert((previousStatus != #closed) || ((currentStatus != #closed)
      && (!isEarlierThan(x, #project)));
  }
}

validator dontChangeEarlierActivities {
  for x in changedQuarcs() {
   assert(!isEarlierThan(previous(x), previous(#project));
  }
}

validator dontMoveToEarlierActivities {
  for x in changed() {
   assert(!isEarlierThan(x, #project);
  }
}
```

```
validator verifyAllClosed {
  if (contains(changedQuarcs(), #project)) {
    for x in changedQuarcs() {
      AttrEnumItem currentStatus;
      currentStatus = getAttributeValueEnumForQuarc(x, #status);
      assert((currentStatus == #closed) || (currentStatus == #dormant)
          || (!isEarlierThan(x,#project)));
    }
  }
}
```

Listing 4.4: QQL program that enforces spiral model characteristics

4.4.2.2 Unified Software Development Process

The Unified Software Development Process [JBR99] (also Unified Process, or UP) is an iterative software life cycle model described by Booch, Jacobson, and Rumbaugh. It evolved from the Rational Unified Process, originally developed by Rational Software. The UP defines four phases: Inception, Elaboration, Construction, and Transition. Each phase consists of one or more iterations. At the end of these four phases, a product is delivered to the stakeholders. One complete set of four phases is a cycle. A UP project can span several cycles, each of which delivers a product. The UP recommends activities and result artifacts for each phase, without defining strict constraints. In each cycle, there are seven workflows performed in parallel: Management, Environment, Requirements, Design, Implementation, Assessment and Deployment. The amount of resources required for each of the workflows varies between phases and iterations (see fig. 4.19). Figure 4.20 shows a quarc repository that represents the mobile dictation software upgrade project in a Unified Process life cycle in iteration 2 of the Construction phase of project cycle 2 (cycle 1 quarcs omitted for brevity).

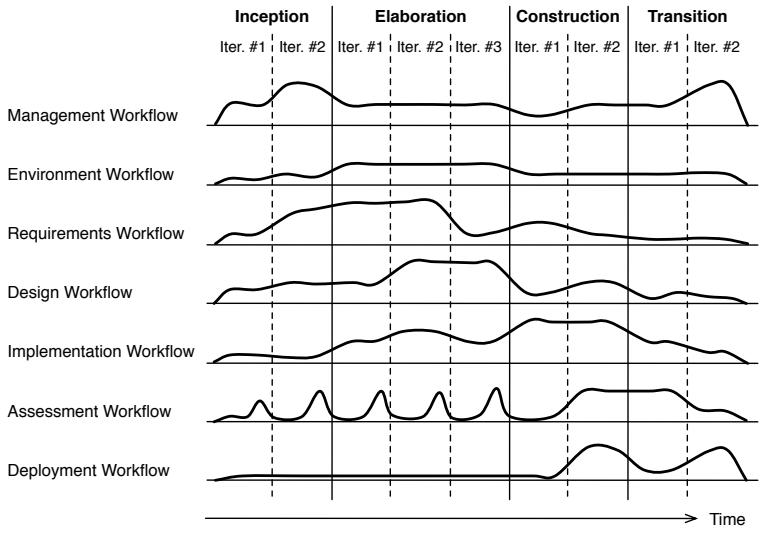


Figure 4.19: The seven workflows in the Unified Process

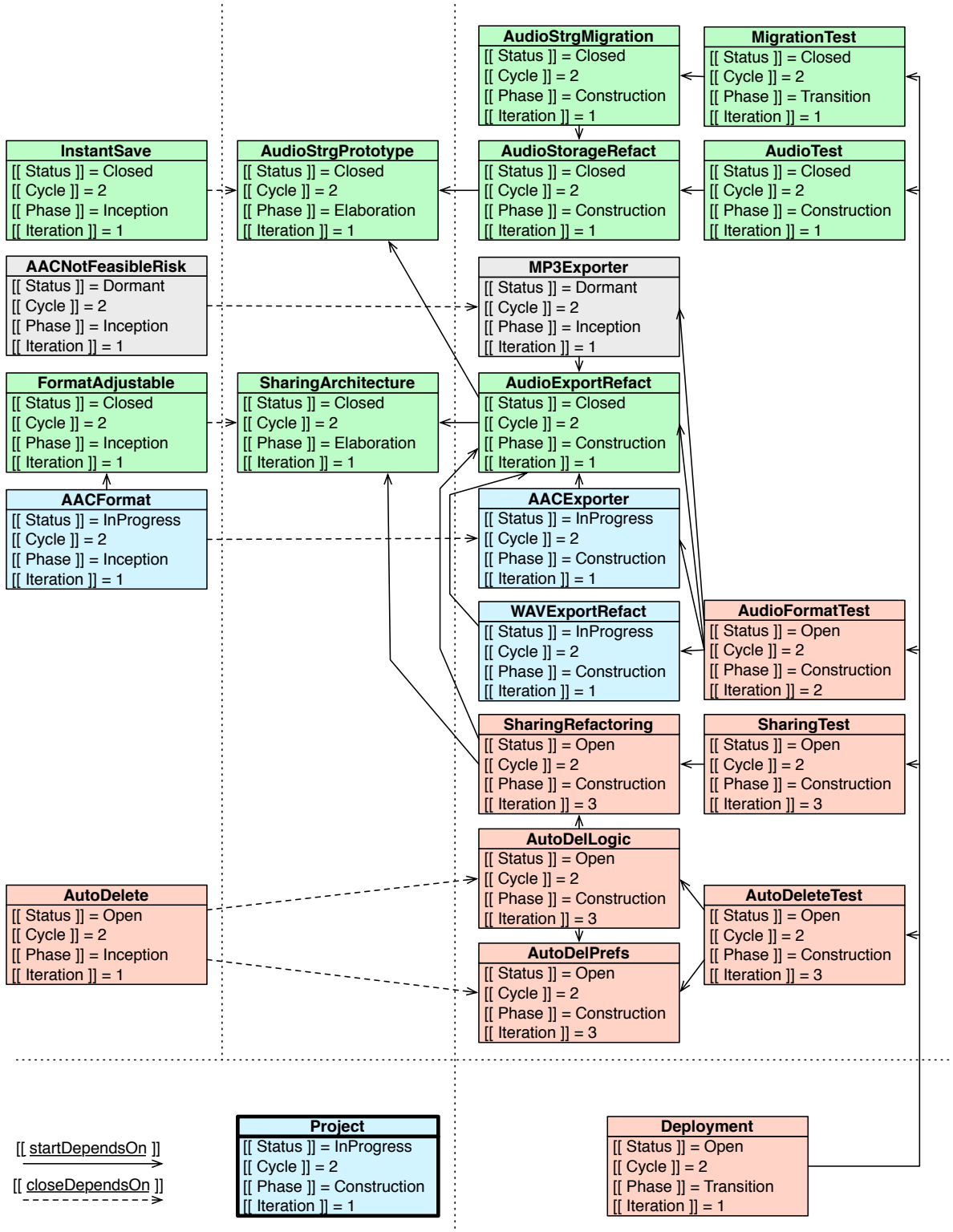


Figure 4.20: Dictamus upgrade project as Unified Process quarcs

This quarc repository is an excerpt from the Dictamus upgrade project modeled as a Unified Process project. The project is currently in cycle no. 2, in the first of three iterations of the Construction phase. The Inception and Elaboration phases are already done, with one iteration each. The next phase will be Transition with one iteration.

A Unified Process project doesn't have the strict order requirement of Waterfall or Spiral Model projects. The only general constraints are that project phases move in a defined order, and that one cycle needs to be finished before the next one starts. We can enforce Unified Process properties in a quarc repository by applying the following predefined quarcs, attributes, and validators to the project.

The master quarc Project represents the entire project. In order to use the master quarc in validators, we define a global variable #project and assign Project to it. Then, we define an attribute projectCycle that represents a project cycle in the quarc graph timeline. ProjectCycle is a numeric attribute that serves as a cycle number. The value for ProjectCycle on the master quarc Project determines the current project cycle. Furthermore, we define an attribute CyclePhase that represents a phase within a project cycle. CyclePhase is an enumerated-type attribute that defines one value for each phase ({Inception, Elaboration, Construction, Transition}). The value for CyclePhase on the master quarc Project determines the current cycle phase. We also define an attribute PhaseIteration that represents an iteration within a project phase. PhaseIteration is a numeric attribute that serves as an iteration number. The value for PhaseIteration on the master quarc project determines the current iteration. We also need to enforce the following structural integrity constraints on quarc repository transactions.

- Closed quarcs cannot be changed anymore. The only exception is reopening a quarc, which means changing its status to another value than Closed or Dormant. If a quarc is reopened, it must be assigned to a later project round, or to the current round and the current or a later activity.
- Quarcs that belong to earlier cycles cannot be changed anymore.
- Quarcs cannot be moved to past cycles.
- A project can only move to the next cycle if all quarcs that are assigned to the current cycle have either status Closed or Dormant.
- The project master quarc may only move sequentially through cycles/phases/iterations.

See listing 4.5 for a representation of these constraints as QQL code.

```
function Bool isEarlierThan(Quarc x, Quarc y) {
   Numeric cycleX = getAttributeValueNumericForQuarc(x, #projectCycle
      );
   Numeric cycleY = getAttributeValueNumericForQuarc(y, #projectCycle
      );
   if (cycleX < cycleY)
      return TRUE;
   if (cycleX > cycleY)
      return FALSE;
   AttrEnumItem phaseX = getAttributeValueEnumForQuarc(x, #cyclePhase
      );
   AttrEnumItem phaseY = getAttributeValueEnumForQuarc(y, #cyclePhase
      );
```

```
   if (isInOrder(phaseX, phaseY))
      return TRUE;
   if (isInOrder(phaseY, phaseX))
      return FALSE;
   AttrEnumItem iterationX = getAttributeValueNumericForQuarc(x, #
      phaseIteration);
   AttrEnumItem iterationY = getAttributeValueNumericForQuarc(y, #
      phaseIteration);
   return iterationX < iterationY;
}

function Bool isEarlierCycleThan(Quarc x, Quarc y) {
   Numeric cycleX = getAttributeValueNumericForQuarc(x, #projectCycle
      );
   Numeric cycleY = getAttributeValueNumericForQuarc(y, #projectCycle
      );
   return cycleX < cycleY;
}

validator dontChangeClosed {
  for Quarc x in changedQuarcs() {
   AttrEnumItem previousStatus;
   AttrEnumItem currentStatus;
   previousStatus = getAttributeValueEnumForQuarc(previous(x), #
      status);
   currentStatus = getAttributeValueEnumForQuarc(x, #status);
   assert((previousStatus != #closed) || ((currentStatus != #closed)
      && (!isEarlierCycleThan(x, #project)));
  }
}

validator dontChangeEarlierCycle {
  for x in changedQuarcs() {
   assert(!isEarlierCycleThan(previous(x), previous(#project));
  }
}

validator dontMoveToEarlierCycles {
  for x in changed() {
   assert(!isEarlierCycleThan(x, #project);
  }
}

validator verifyAllClosed {
  if (contains(changedQuarcs(), #project)) {
   for x in changedQuarcs() {
    AttrEnumItem currentStatus;
    currentStatus = getAttributeValueEnumForQuarc(x, #status);
```

```
    assert((currentStatus == #closed) || (currentStatus == #dormant)
        || (!isEarlierCycleThan(x,#project)));
   }
  }
}

validator projectMovesSequentially {
  assert(!isEarlierThan(#project,
   previous(#project)));
}
```

Listing 4.5: QQL program that enforces Unified Process characteristics

4.4.3 Agile Models

Scrum [SB01] is an iterative agile software development methodology. It is based on the notion that modern software projects are too complex to be fully planned in the early project stages. The Scrum concept was first mentioned in a 1986 article by Takeuchi and Nonaka in Harvard Business Review [TN86], where they described a new "holistic" method. They compare it to a ball getting passed in a rugby team moving down a game field. They elaborated it later in their book "The Knowledge-Creating Company" [NT95]. Schwaber used this approach in his company Advanced Development Methods. Sutherland and others developed a similar approach at Easel Corporation. They were the first to call it "Scrum". In 1995, Schwaber presented a paper describing the Scrum methodology at a workshop at OOPSLA 95 [Sch96].

In the beginning of a Scrum project, the product owner collects everything that should be done in the project (requirements, bug fixes, etc) and compiles it into a prioritized list of items. This list is called *product backlog*. The actual project work is performed in time-boxed iterations, *sprints*. The goal is to produce a (potentially shippable) working product increment after each sprint. The length of a sprint is usually a few weeks, and it is recommended that sprints are always the same length.

At the beginning of each sprint, the team determines which items from the product backlog will be implemented. The team picks the highest-priority item from the product backlog, estimates how long it will take to implement, places it in the sprint backlog for the new sprint, and calculates how much time is left in the sprint. This process is repeated until no more items fit into the remaining time. Next, the development team breaks up each item in the sprint backlog into tasks and decides how to distribute the work. During the sprint, the entire team meets briefly each morning ("Daily Scrum"), where each member reports what he/she did, what he/she intends to do today, and if there are any blockers or impediments. A typical Scrum team has seven plus/minus two members. See figure 4.21 for a depiction.

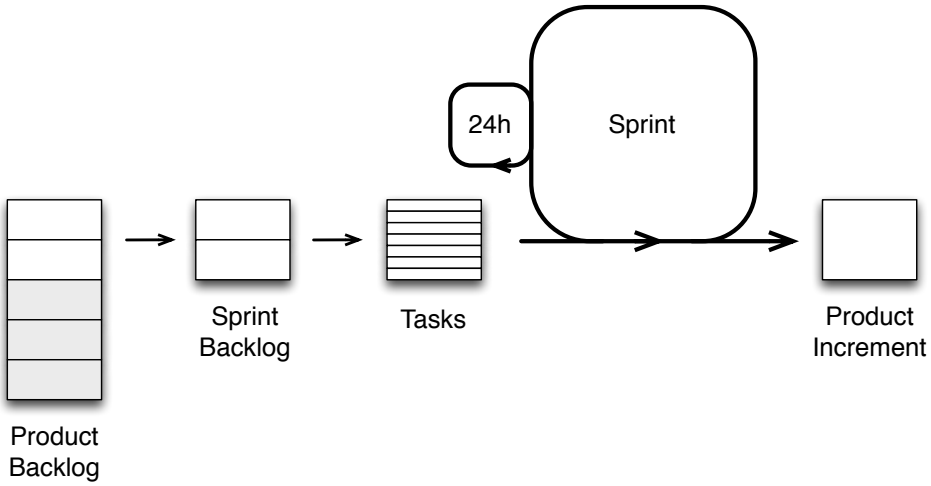


Figure 4.21: The Scrum development process (adapted from [SB01])

Figure 4.22 shows a quarc repository that represents the Dictamus upgrade project in a Scrum organization. Each item is represented by a quarc.

There are two approaches to enforce Scrum properties in a quarc repository. The first is to represent sprints by an enumerated-type or numeric attribute Sprint, the second is to introduce one quarc for each sprint backlog and the product backlog. Common to both approaches is the representation of each item by one quarc, without any additional attributes. The attribute approach is easier to enforce in validators because each quarc carries the relevant information in an attribute, whereas in the backlog quarc approach, each quarc must be traced to its backlog quarc. Furthermore, in the backlog quarc approach, it is possible that a non-backlog quarc is linked to multiple backlog quarcs through dependencies. On the other hand, in the attribute approach, it is necessary to walk the entire dependency graph and change all dependent quarcs' Sprint attributes to move an item from one sprint backlog to another. In the backlog quarc approach, just one dependency of the item quarc is changed, i.e. moved from old backlog quarc to new backlog quarc. The attribute approach is similar to the concepts we presented for the life cycle models in the previous sections, so we describe the backlog quarc approach instead to give a broader impression of the metamodel's capabilities.

The relations we establish prevent a sprint from being closed when there are still unclosed items on which it depends hierarchically (closeDependsOn). The Quarc repository enforces this as structural integrity constraints and aborts the status change on the sprint quarc. In an actual project, there may be situations where a project manager wants to finish a sprint (which translates to setting the sprint quarc's status to closed) although it has unfinished items. An actual QUARC-based CASE tool that supports Scrum could simply refuse to perform this operation and point out the quarc relations and quarcs that caused the abort, but it would be preferable for it to offer intelligent solutions to accommodate the operation instead. The CASE tool can, for example, assume that all quarcs to which a sprint has a closeDependsOn quarc relation represent items, and offer the project manager to move all unclosed items from that sprint back to the product backlog, or create a new sprint and move them into that. On the repository layer, this

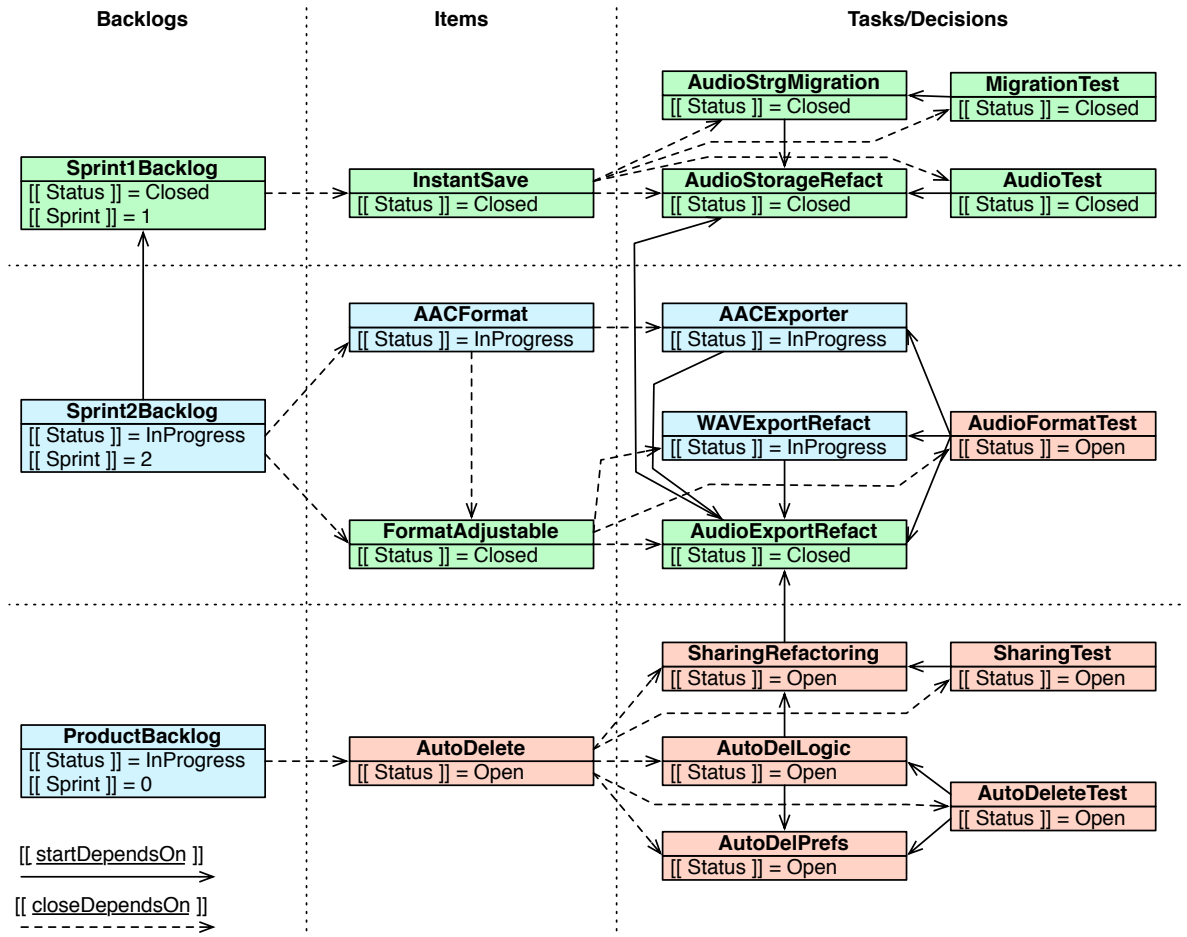


Figure 4.22: Dictamus upgrade project as Scrum quarcs

This excerpt from a quarc repository that represents the Dictamus upgrade project as a Scrum project shows the product backlog with one remaining item, the finished sprint 1 with one item, and sprint 2 which is currently in progress, with one finished item and one item in progress.

means that the tool removes the quarc relations that block the status change and creates new ones to the product backlog quarc or new sprint quarc, respectively.

We define a backlog quarc ProductBacklog that represents the product backlog. In order to use it in validators, we define a global variable #productbacklog and assign ProductBacklog to it. Furthermore, each time a new sprint is started, we create a new backlog quarc for that sprint. The current sprint backlog quarc is stored in global variable #currentsprint. Then, we define the numeric attribute Sprint to mark backlog quarcs.

Backlog quarcs have a value assigned to their Sprint attribute that indicates their sprint number. ProductBacklog's Sprint value is zero. All regular quarcs (i.e. quarcs that are not backlog quarcs) do not have any value assigned to their Sprint attributes. We also define constraints that ensure there is only one product backlog quarc, and only one sprint quarc that is not closed. A

representation of these constraints as QQL code is shown in listing 4.6.

```
validator onlyOneProductBacklog {
  Bool found;
  found = FALSE;
  for Quarc x in quarcs() {
    if (isAttributeValueAssignedToQuarc(x, #sprint)) {
      Numeric sprintnum;
      sprintnum = getAttributeValueNumericForQuarc(x, #sprint);
      if (sprintnum == 0) {
        assert(!found);
        found = TRUE;
      }
    }
  }
}

validator onlyOneOpenSprint {
  Bool found;
  found = FALSE;
  for Quarc x in quarcs() {
    if (isAttributeValueAssignedToQuarc(x, #sprint)) {
      Numeric sprintnum;
      sprintnum = getAttributeValueNumericForQuarc(x, #sprint);
      AttrEnumItem status;
      status = getAttributeValueEnumForQuarc(x, #status);
      if ((sprintnum > 0) && (status != #closed)) {
        assert(!found);
        found = TRUE;
      }
    }
  }
}
```

Listing 4.6: QQL program that enforces Scrum characteristics

The closeDependsOn dependencies from sprint backlog quarcs to their items prevent a sprint quarc from being closed while there are unclosed items. Listing 4.7 shows a QQL program that creates a new sprint, moves unfinished items to the product backlog or the new sprint, and closes the old sprint.

```
function Quarc closeCurrentAndStartNewSprint(Bool
    moveUnfinishedToBacklog) {
    Quarc lastsprint = #currentsprint;
    Numeric lastsprintnum = 0;
    List unfinished;
    unfinished = createListQuarc();
    if (lastsprint != nil) {
      // Find unfinished quarcs
```

```
    List dep;
    dep = quarcsRelatedFromQuarc(lastsprint, #closeDependsOn);
    for Quarc x in dep {
      AttrEnumItem status_rel;
      status_rel = getAttributeValueEnumForQuarc(x, #status);
      if (status_rel != #closed) {
        // Remove from old sprint and keep in local list
        appendToList(unfinished, x);
        disestablishQuarcRelation(#closeDependsOn, lastsprint, x);
      }
    }
    // Close sprint
    setAttributeValueEnumForQuarc(lastsprint, #status, #closed);
    lastsprintnum = getAttributeValueNumericForQuarc(lastsprint, #
       sprint);
  }

  Quarc newsprint = createQuarc();
  assignAttributeValueToQuarc(newsprint, #sprint, lastsprintnum+1);
  assignAttributeValueToQuarc(newsprint, #status, #inProgress);
  if (lastsprint != nil) {
    establishQuarcRelation(#startDependsOn, newsprint, lastsprint);
  }
  #currentsprint = newsprint;

  // Move unfinished quarcs to backlog or new sprint
  Quarc unfinishedTarget;
  if (moveUnfinishedToBacklog) {
    unfinishedTarget = #productbacklog;
  } else {
    unfinishedTarget = newsprint;
  }
  for Quarc x in unfinished {
     establishQuarcRelation(#closeDependsOn, unfinishedTarget, x);
  }

  return newsprint;
}
```

Listing 4.7: Start a new sprint and move unclosed issues

During a Scrum sprint, a common progress indicator is the *Burndown Chart*. A burndown chart plots the number of unfinished tasks over time since the start of the sprint. Ideally, the plot is monotonically decreasing down to zero when the sprint reaches its end. Usually, there will be spikes upwards due to unforeseen complications, and the plot won't reach zero because some open issues remain at the end of the time-boxed sprint. These are moved back to the product backlog.

To obtain the underlying metrics for a burndown chart from a Scrum project modeled as a quarc repository, we proceed as follows: The number of unfinished tasks can be counted by

traversing the quarc dependency hierarchy starting at the sprint backlog quarc, and counting quarcs that are in progress or open. To obtain the remaining effort in hours as in the sample chart, we use the default Duration attribute.

4.5 Dynamic Tailoring

In sections 4.4.1 and 4.4.3, we showed that the QUARC metamodel can represent Waterfall and Scrum lifecycles. A Waterfall lifecycle in a quarc repository is based on the same entity as a Scrum lifecycle: the quarc. In this section, we explore how a Waterfall quarc repository can be transitioned into a Scrum quarc repository without rebuilding the entire model from scratch.

Figure 4.23 shows the Dictamus upgrade project in a Waterfall lifecycle from subsection 4.4.1 in an earlier project phase (Implementation).

At this point, the project manager wants to switch to a Scrum lifecycle to deal with a change in requirements or technology. The design phase quarcs in the Waterfall model are already similar to product backlog items, so the most straightforward way to handle the transformation from the Waterfall quarc repository to a Scrum quarc repository is to convert the design quarcs to product backlog item quarcs. The transformation process for the Waterfall quarc repository is structured as follows:

1. Remove any Waterfall-specific validators
2. Remove the Project quarc
3. Remove the Requirements Elicitation quarc and all its relations
4. Perform these steps in a single transaction to avoid integrity constraint violations
   (a) Make a list of all quarcs with Phase = Design ("backlog item quarcs")
   (b) Change the status of all backlog item quarcs to Open
   (c) For each backlog item quarc, walk the quarc graph from the backlog item quarc in reverse direction along the startDependsOn quarc relation, and create closeDependsOn quarc relations from the backlog item quarc to each visited quarc. In other words, establish a parent/child relation from the backlog item quarc to all quarcs in its transitive closure on the reversed startDependsOn quarc relation.
   (d) Remove all startDependsOn quarc relations that end on a backlog item quarc
5. Remove assigned values for the Phase attribute from all quarcs
6. Remove the Phase attribute type
7. Add the Sprint attribute type
8. Add the ProductBacklog quarc, with status InProgress and value 0 assigned for Sprint.

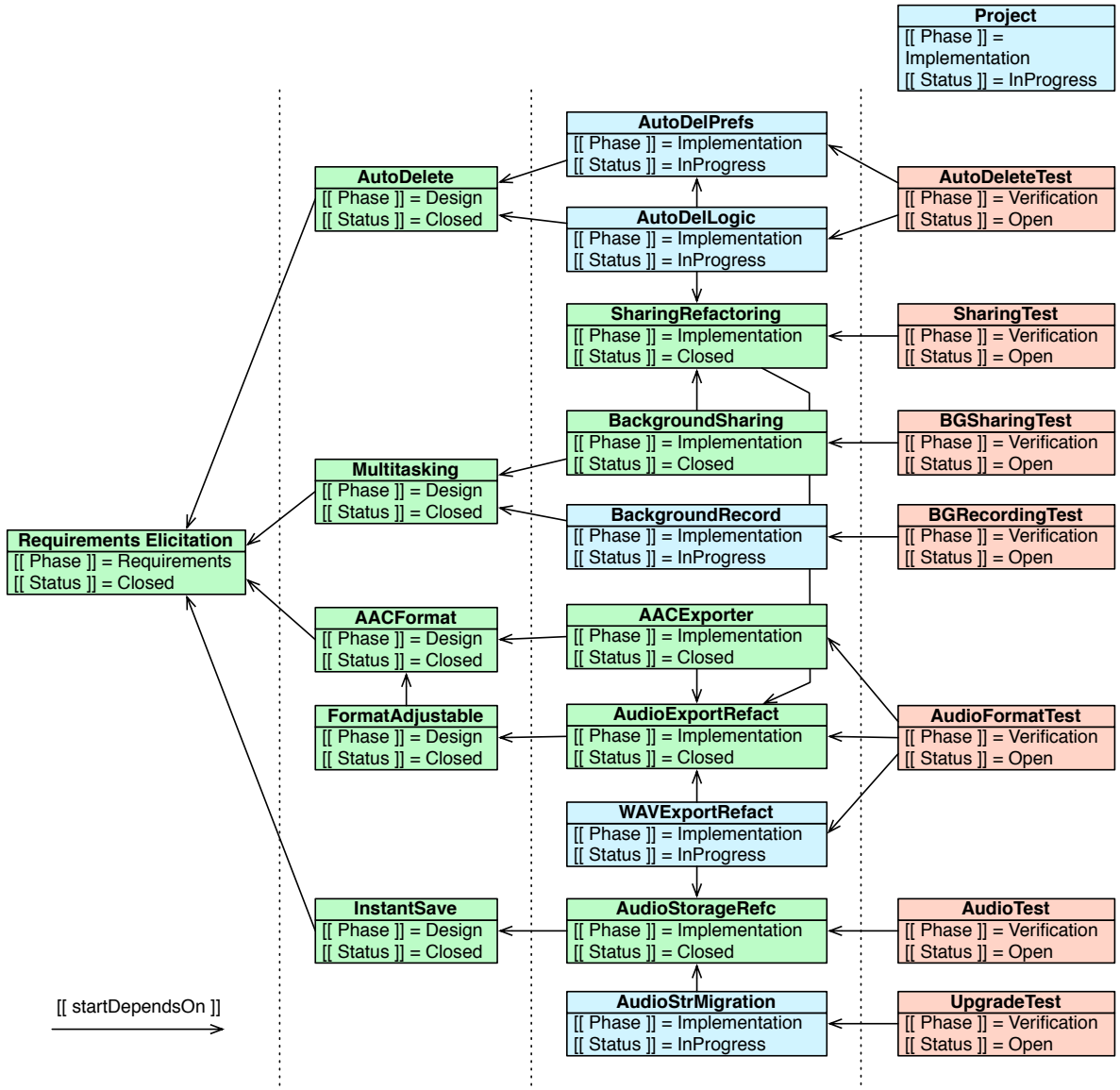


Figure 4.23: Dictamus upgrade project in a Waterfall software life cycle

9. Create closeDependsOn quarc relations from the ProductBacklog quarc to all backlog item quarcs

10. Add Scrum-specific validators

This process can be fully automated if the Waterfall quarc repository is structured like the one in figure 4.23. Figure 4.24 shows the result of this process applied to the repository in figure 4.23. After the transformation, the project manager can start the Scrum process to create and launch the first sprint.

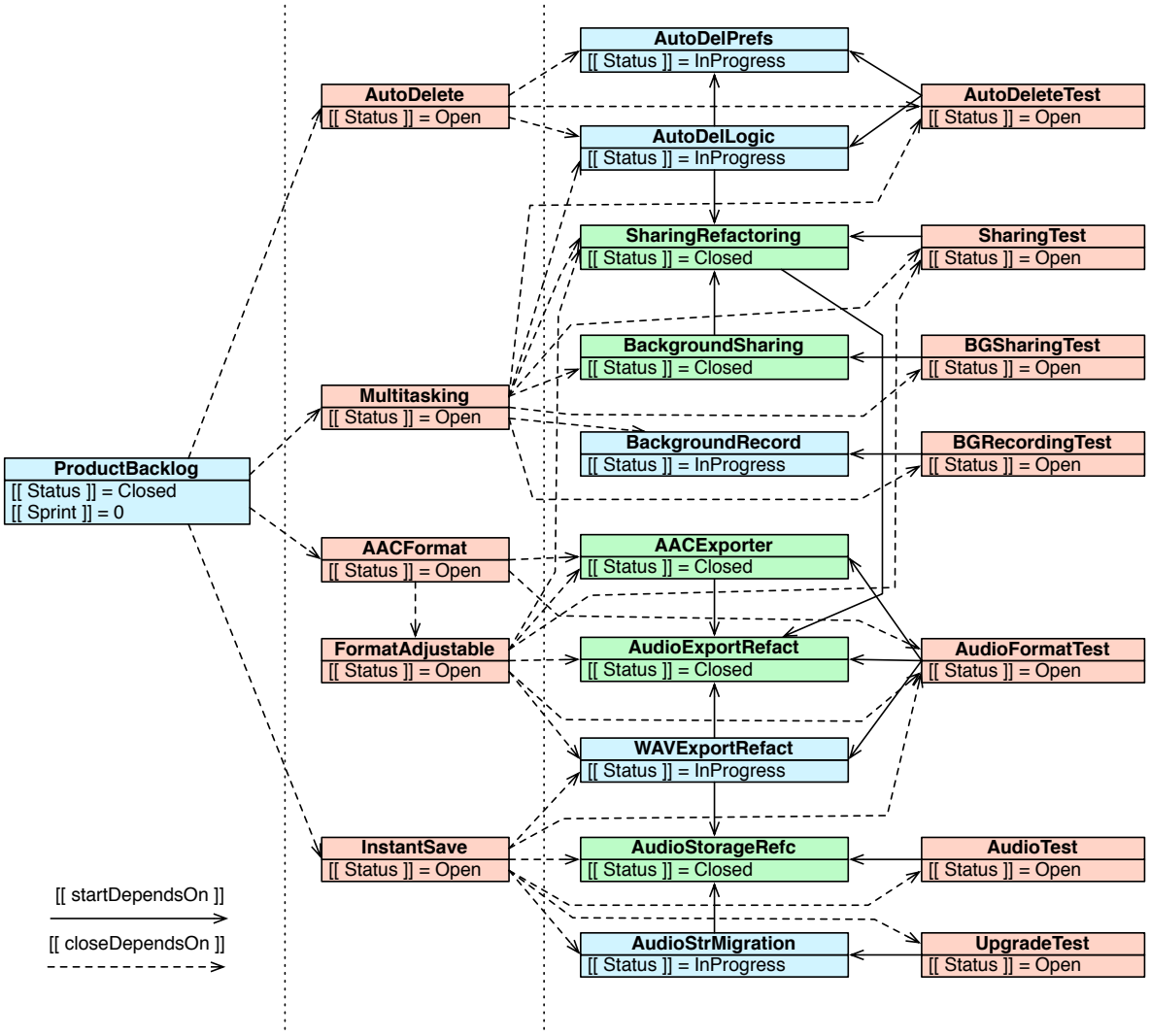


Figure 4.24: Dictamus upgrade project after transformation from Waterfall to Scrum

4.6 Hybrid Models

We showed quarc repository models for task management, issue management and risk management. The default quarc repository model (see subsection 2.6.3) can be extended to form a hybrid task/issue/risk management quarc repository model, as shown in figure 4.25. We added the attribute Critical from the PERT/CPM repository model (see section 4.2) to the default quarc repository.

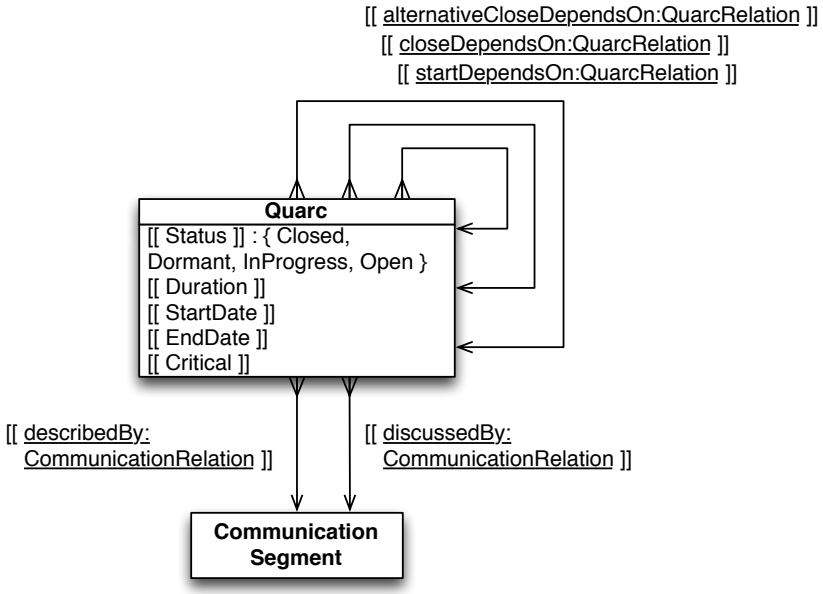


Figure 4.25: Hybrid task/issue/risk management quarc repository model
This quarc repository model is an extension of the default quarc repository model to support task management, issue management and risk management in a single hybrid model.

In figure 4.26, we show an instance of the hybrid model for the Dictamus upgrade. For brevity reasons, we only show one decision and one risk to illustrate the hybrid nature of the model, and we omit the StartDate and EndDate attributes. The group in the upper left that consists of the WhichFormat, AAC, Speex, MP3, and Exporter quarcs represents a decision that handles the question "Which audio format should we support". WhichFormat represent the task of making this particular decision. Exporter is a "parent" quarc for the three possible solutions, AAC, Speex, and MP3. It has a alternativeCloseDependsOn quarc relation to each of them, which indicates that Exporter cannot be closed before one of the three solutions is closed. The risk StorageRisk, located in the bottom left corner, represents the risk that the file storage implementation in the project violates new platform guidelines that are announced, but not yet available. The quarc ReengineerStorage represents the risk mitigation activity that is assigned to this risk.

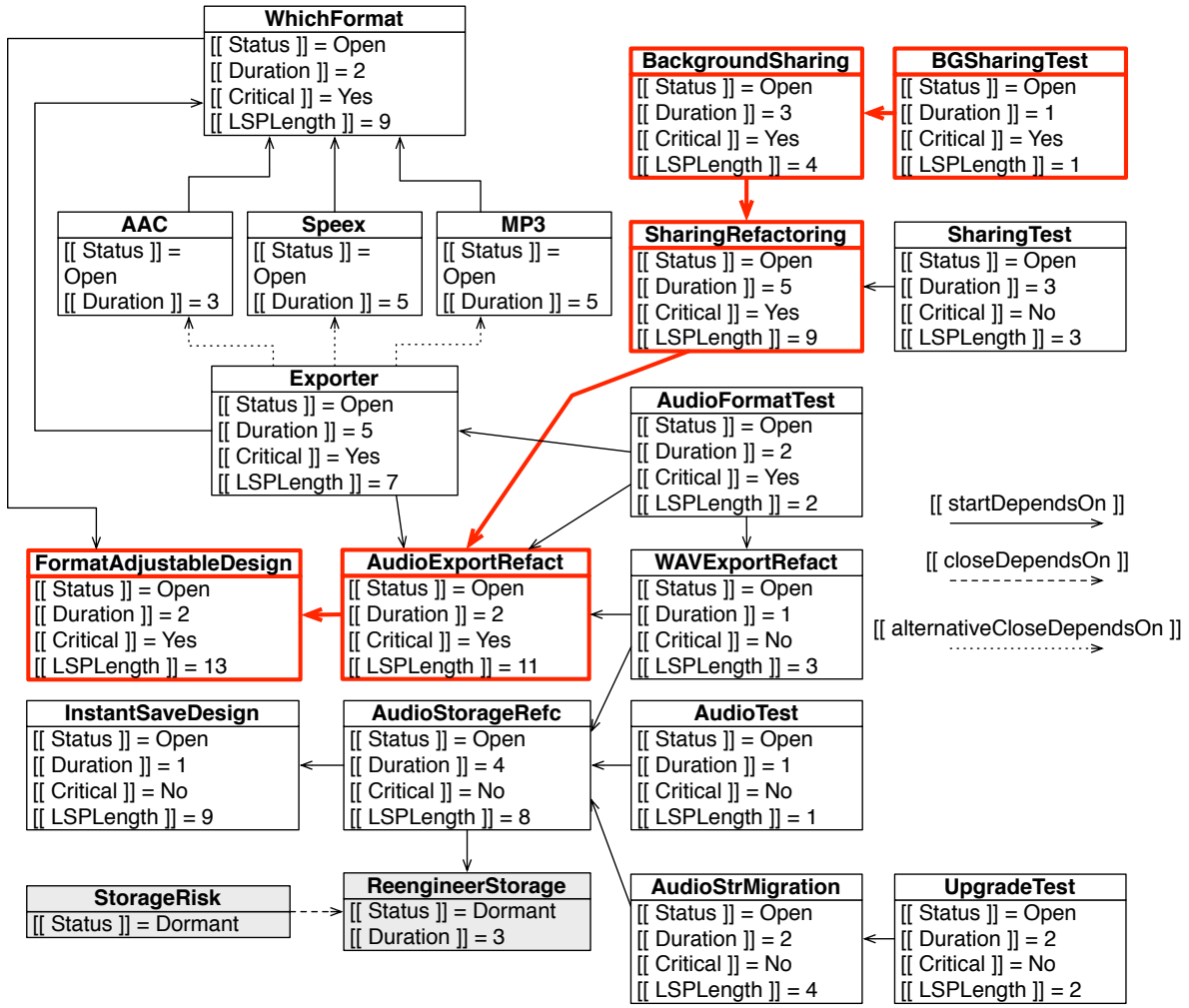


Figure 4.26: Hybrid task/issue/risk management project (day 1)

This quarc repository shows the Dictamus project as a task/issue/risk hybrid with one decision and one risk. The critical path is shown in red lines (thicker lines in black & white).

The diagram in figure 4.26 also shows the current critical path. As shown in section 4.2, we can set up a validator that updates the critical path and the longest sub path lengths for all quarcs after each transaction. These lengths are assigned to the LSPLength attribute on each quarc.

Figure 4.27 shows the state of the project on day 3. The WhichFormat decision is in progress. The availability of player software for each of the three formats has been evaluated, and the results were recorded as communication artifacts and communication segments in the repository (represented by a document symbol). Software availability for AAC and Speex was not as good as for MP3, so AAC and Speex were connected to the "Not widely available" communication segment. MP3 was connected to the "Widely available" communication segment.

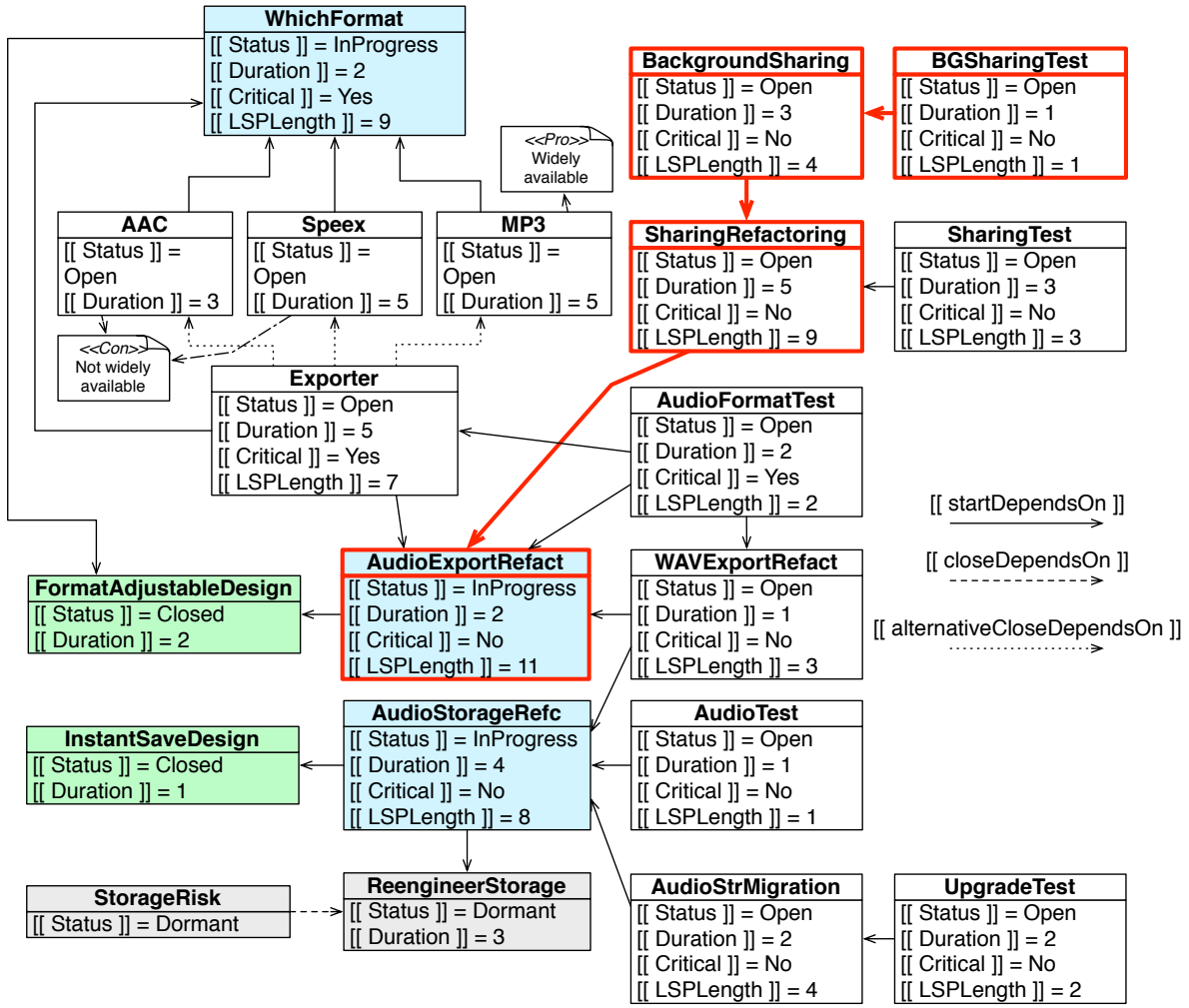


Figure 4.27: Hybrid task/issue/risk management project (day 3)

The decision between audio formats is currently in progress. The availability of player applications was evaluated, and the results were recorded as communication artifacts and segments (document symbols in the diagram).

Figure 4.28 shows the state of the project on day 5. The implementation effort for the three formats has been evaluated, and two new communication artifacts and segments were added. AAC was connected to "Easy to implement", Speex and MP3 were connected to "Hard to implement". During this evaluation, it became apparent that the initial estimate of 5 days for the duration of the MP3 implementation was too optimistic. The estimate was raised to 9 days. This change is reflected in the Duration attribute value in the quarcs MP3 and Exporter. Afterwards. the WhichFormat decision was made, and MP3 was selected despite its high implementation effort. As a result, the critical path has changed; it now includes the Exporter quarc.

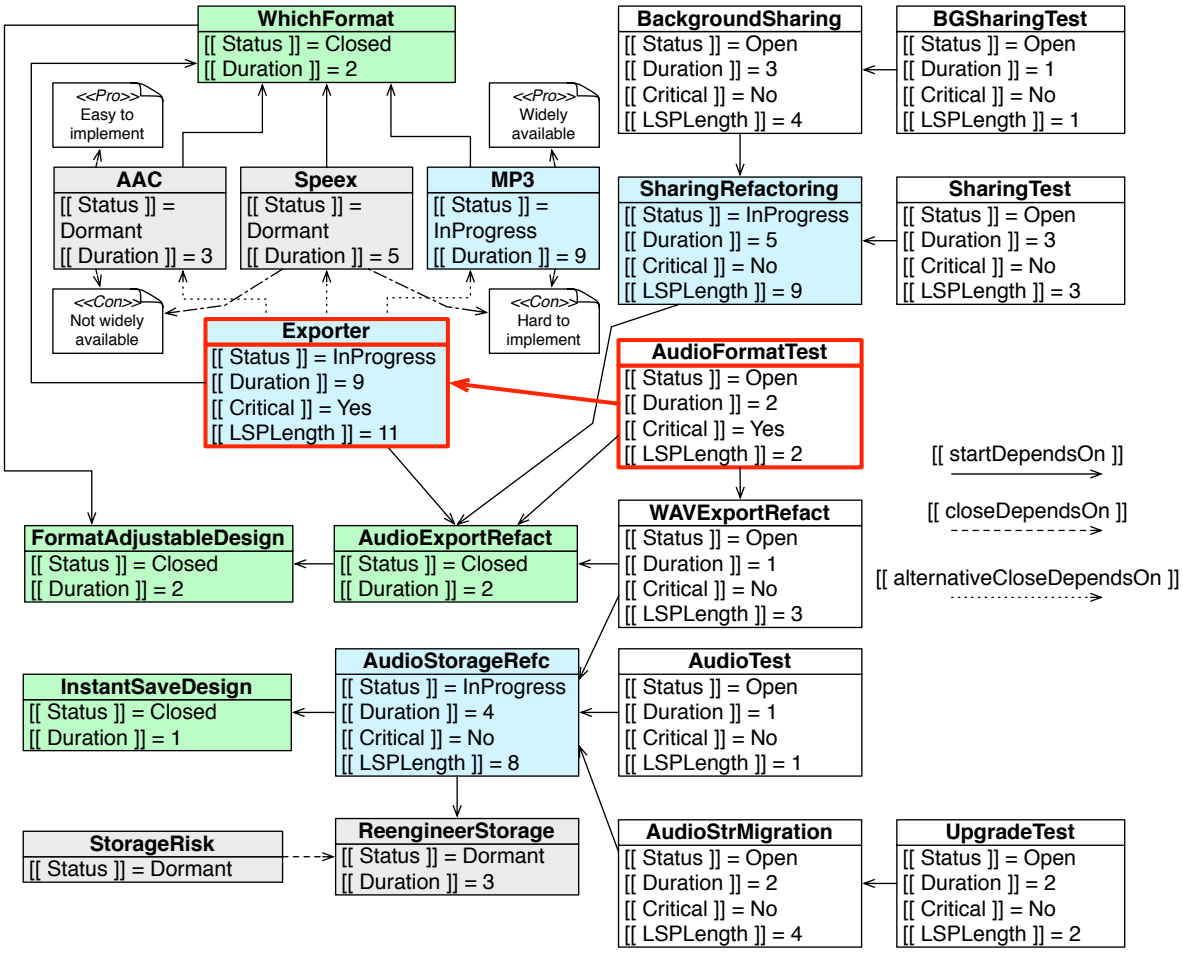


Figure 4.28: Hybrid task/issue/risk management project (day 5)

The decision between audio formats has been made in favor of MP3. The amount of effort required for the implementation of the three formats was evaluated, and the results were again recorded as communication artifacts and segments. The estimated duration of the MP3 quarc and the Exporter quarc had to be raised from 5 to 9 days. This caused a change in the critical path, which now includes the Exporter.

Figure 4.29 shows the state of the project on day 14. The Exporter task was finished in the meantime, but the SharingRefactoring task was neglected. As a result, the critical path has reverted to its state from the beginning, minus the tasks that were finished in the meantime.

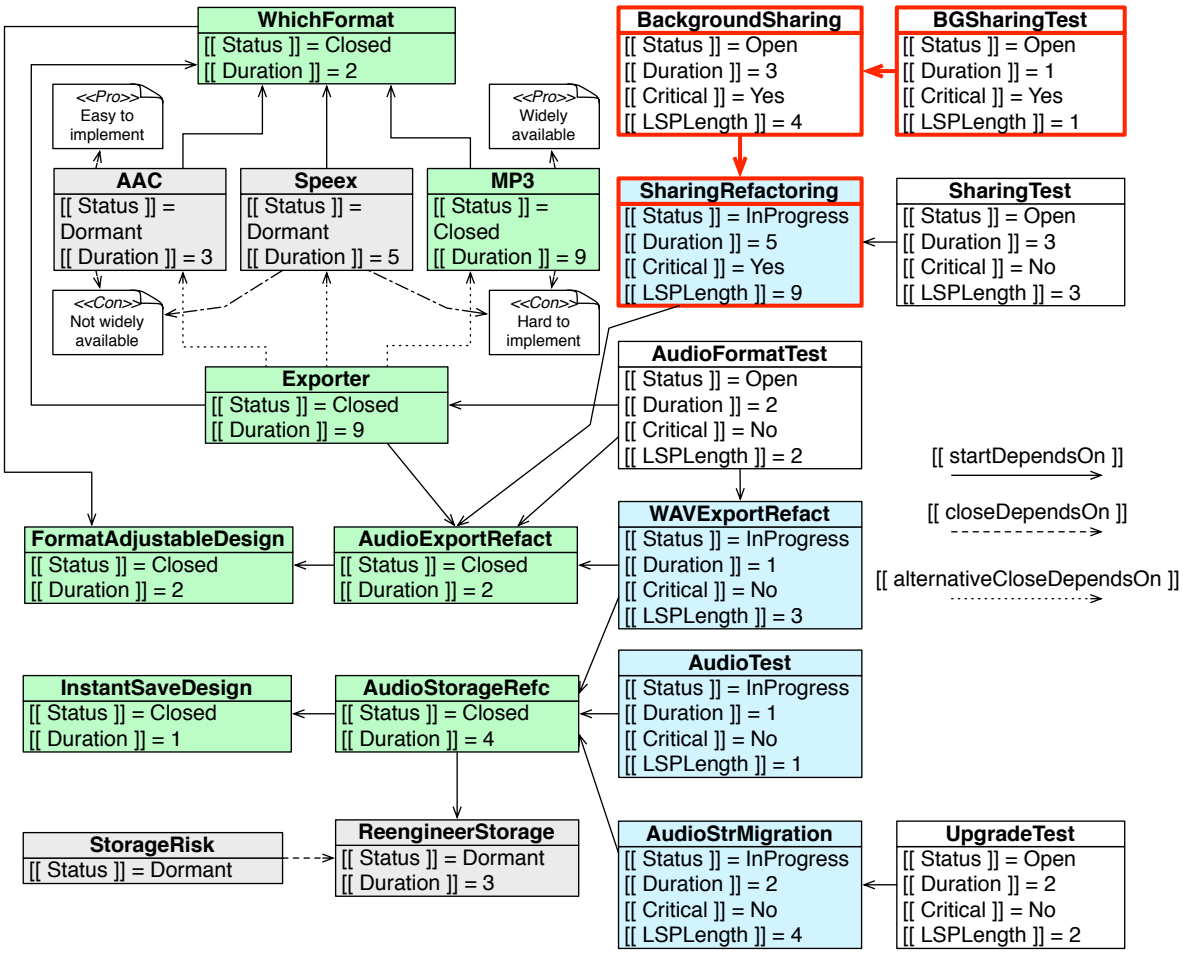


Figure 4.29: Hybrid task/issue/risk management project (day 14)

The Exporter task has been finished, and the critical path reverted to the earlier one.

Figure 4.30 shows the state of the project on day 15. The StorageRisk risk has materialized, and the ReengineerStorage task was opened and started as a consequence. As described in section 4.3, the following quarcs need to change to status Open as well to satisfy structural integrity constraints: AudioStorageRefc, WAVExportRefact, AudioTest, and AudioStrMigration. As a result, the critical path changed again and now includes ReengineerStorage.

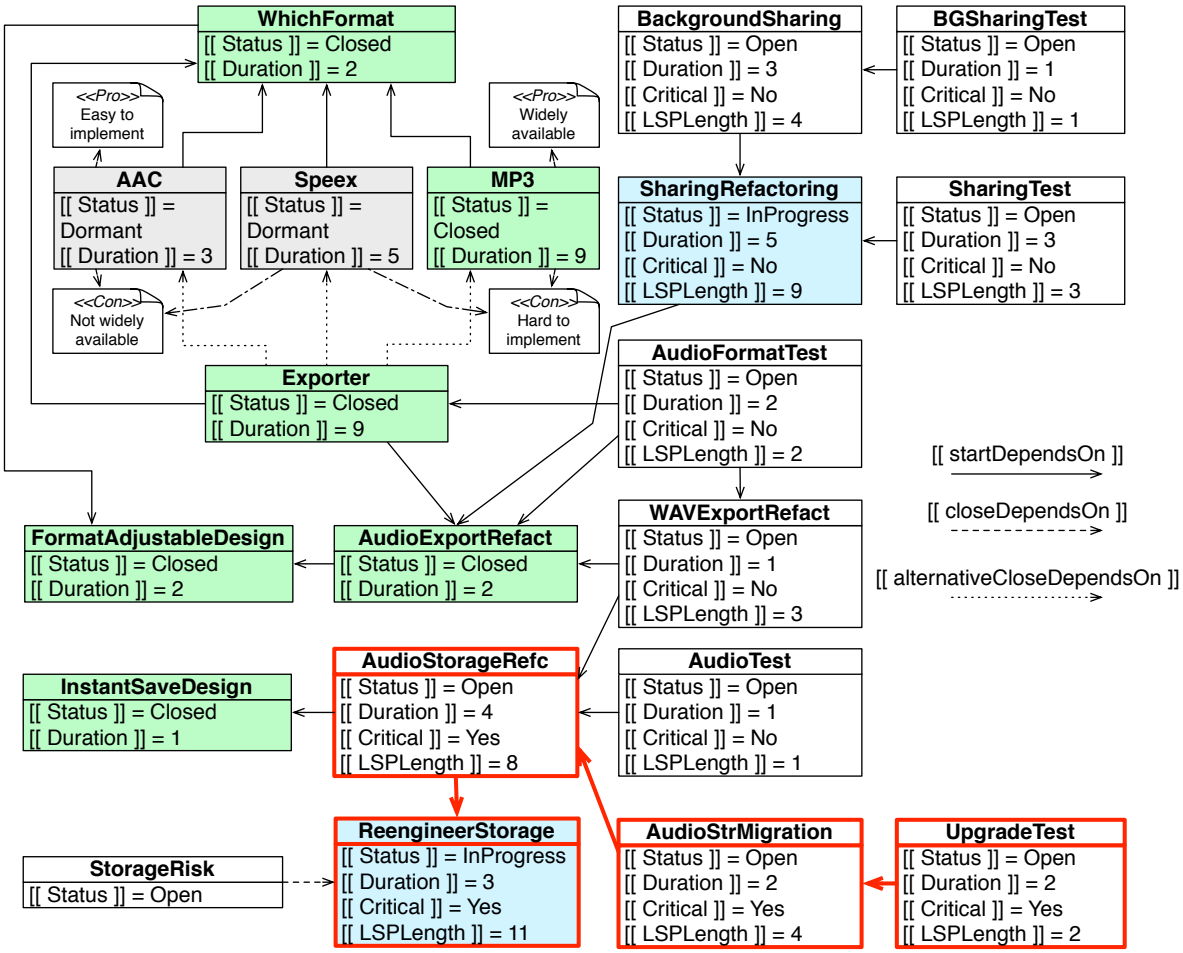


Figure 4.30: Hybrid task/issue/risk management project (day 15)

The risk Storage has materialized, and the risk mitigation measure ReengineerStorage was opened as a consequence. This forced the quarcs AudioStorageRefc, WAVExportRefact, AudioTest, and AudioStrMigration ro reopen as well.

CHAPTER 5

Conclusion

5.1 Contributions

In this dissertation, we introduced the QUARC metamodel as a unified model for issue management, task management, risk management and software lifecycle. We showed how the QUARC metamodel incorporates human-to-human communication such as voice recordings as first-class citizens in its model. Furthermore, we presented the Quarc Query Language (QQL) as a universal language to query and modify a quarc repository, and to define constraints. We showed how widely used models for issue management, task management and software lifecycle can be expressed in quarc repositories, and we introduced hybrid models that combine aspects from these models. We have shown that the QUARC metamodel is a generalization of many existing lifecycle models. In particular, we have shown how activity-oriented models such as Waterfall, V-Model, Spiral Model and the Unified Software Development Process as well as entity-oriented lifecycle models such as Scrum and XP can be represented as QUARC models.

With QQL, we form a new expressive language that allows operations on quarc repositories, similar to what SQL is for relational databases. In QQL, we express the validators that define the structural integrity constraints of a quarc repository. We showed how specific integrity constraints can impose a specific software lifecycle model on the quarcs in a quarc repository. Due to the flexible nature of QQL, arbitrary lifecycle models can be expressed by placing appropriate QQL validators and administration quarcs in a quarc repository.

Going even further beyond classic lifecycle models, QUARC allows dynamic tailoring of projects and processes. By modifying the validators in a quarc repository at project runtime, an organization can change the behavior of projects and processes while they are in progress. We showed a process that can convert a Waterfall-oriented quarc repository to a Scrum-based one. Due to the common underlying entity model, quarcs, it is possible to transform a project's lifecycle model by replacing the set of validators that enforces one particular model with another, and by making some structural adjustments which may be partially automated.

5.2 Future Work

We have laid the foundation for a new project metamodel. We propose an experimental validation of the quarc concept in incremental studies, going from small projects with a narrow scope to big ones. For any kind of practical experiment or case study on the QUARC metamodel, we expect that it will be necessary to implement a QUARC foundation that allows CASE tools to be built on top of it, or to replace the entity layer of existing CASE systems, partially or entirely.

As soon as a QUARC-based CASE tool is available, further studies can be performed about the aspect of including human-to-human communication in CASE systems. Such a CASE tool would improve the tool support and integration aspects of our initial studies, tool support and integration, which can be rectified by a CASE tool that incorporates the communication aspects of the QUARC metamodel in an easily accessible and natural way.

One of the next research projects could be the investigation of a fast switch capability of the lifecycle model during an ongoing project. With QUARC, we have laid the foundation for support of rapid switches from activity based models such as Waterfall to entity based lifecycle models such as Scrum. Many organizations follow Waterfall or another linear model, and may benefit from such an option. This capability hasn't been available in existing CASE tools, and the impact and feasibility of such switches would have to be investigated.

Another research area is the dynamic tailoring of processes, especially business processes. Common business process notations and languages such as BPMN require exceptions to be defined at design time, they do not support runtime changes to deal with unforeseen events. A QUARC-based business process engine allows the adjustment of business processes at runtime due to the dynamic tailoring capabilities of the QUARC metamodel.

One could also apply the QUARC concept to antipatterns [BMMM98]. Any of the patterns in [BMMM98] can be seen as an issue for which two proposals exist: the currently practiced, refactored solution and an outdated solution that is bad, even though it may historically have been good.

Quarc repositories are currently designed to operate as a centralized single authoritative resource. It would be interesting to apply software configuration management techniques and methods to a quarc repository, possibly with local repository copies for project participants and a branching/merging system. Kögel already investigated SCM for unified models [Kög08] and model merging [Kög11]. His results could probably be adapted to QUARC models.

An extension of the quarc repository concept and QQL to support simulations would enable quarc-based decision support systems. In such a system, a decision maker could study the impact of operations on a quarc repository. In section 4.3, we showed the wave of reopening quarcs when a risk mitigation quarc is opened. A quarc decision support system could show operations like this as an animation, without actually performing any changes to the repository. This allows the visualization of the potential risks and the ramifications of mitigation measures that go into effect when the risks materialize. QUARC could also be used for user-defined

metrics in decision support systems, where a metric is a numeric value calculated on a quarc repository. For example, the number of open quarcs could be a metric showing the progress of the project. Another metric could be the total duration of tasks on the current critical path. The decision support system could display such metrics in its simulations to allow a numerical analysis of the impact of quarc repository changes.

We speculate that it should even be possible to have a limited conversation with a QUARC-based CASE tool by employing limited-vocabulary speech recognition technology to recognize predefined commands, and by using text-to-speech technology to read out an answer from the model. For example, a project manager could ask the QUARC-based tool to "List all open quarcs in the current project phase", and the system would read out or play the summaries of the requested quarcs. We dub this *Hands-Free Software Engineering*.

Bibliography

[AISJ77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, and Max Jacobson. A Pattern Language. Oxford University Press, USA, 1977.

[All02] David Allen. *Getting Things Done: The Art of Stress-Free Productivity*. Penguin Books, first edition, December 2002.

[Anoa] Mantis Bug Tracker [online]. Available from: http://www.mantisbt.org/.

[Anob] UNICASE project [online]. Available from: http://www.unicase.org.

[BBD01] Andreas Braun, Bernd Bruegge, and Allen H Dutoit. Supporting Informal Requirements Meetings. In *7th International Workshop on Requirements Engineering: Foundation for Software Quality. (REFSQ'2001)*, 2001.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development [online]. 2001. Available from: http://agilemanifesto.org/.

[BCCP01] Sumit Basu, Tanzeem Choudhury, Brian Clarkson, and Alex Pentland. Towards measuring human interactions in conversational settings. In *IEEE CVPR Workshop on Cues in Communication*, 2001.

[BCH07] Bernd Bruegge, Oliver Creighton, and Jonas Helming. Unicase - an ecosystem for unified software engineering research tools. *Third IEEE International Conference on Global Software Engineering*, 2007.

[BD10] Bernd Bruegge and Allen H Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java*. Prentice Hall, 3rd edition, 2010.

[BDHB02] Andreas Braun, Allen H Dutoit, Andreas G Harrer, and Bernd Bruegge. IBistro: a learning environment for knowledge construction in distributed software engineering courses. In *Ninth Asia-Pacific Software Engineering Conference, 2002.*, pages 197–203. IEEE Computer Society, 2002.

[BDW06] Bernd Bruegge, Allen H Dutoit, and Timo Wolf. Sysiphus: Enabling informal collaboration in global software development. In *International Conference on Global Software Engineering (ICGSE'06)*, pages 139–148. IEEE Computer Society, 2006.

[BMMM98] William J Brown, Raphael C Malveau, Hays W Skip McCormick, and Thomas J Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1st edition, April 1998.

[Boe86] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, August 1986.

[Boe89] Barry W. Boehm. *Software Risk Management*. IEEE Computer Society Press, August 1989.

[Boe91] Barry W. Boehm. Software Risk Management: Principles and Practices. *IEEE Software*, 8(1):32–41, 1991.

[BR89] Barry W. Boehm and Rony Ross. Theory-W software project management principles and examples. *IEEE Transactions on Software Engineering*, 15(7):902–916, 1989.

[CB88] Jeff Conklin and Michael L Begeman. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. In *ACM conference on Computer-supported cooperative work (CSCW'88)*, pages 140–152, New York, New York, USA, 1988. ACM Press.

[Con68] Melvin E Conway. How Do Committees Invent? *Datamation*, 14(4):28–31, 1968.

[CR09] Sebastien Cherry and Pierre N Robillard. Audio-Video Recording of ad hoc Software Development Team Interactions. In *ICSE Workshop on Cooperative and Human Aspects on Software Engineering (CHASE'09)*, pages 13–21, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[CWH99] Michael G Christel, Howard D Wactlar, and Alexander G Hauptmann. Informedia Digital Video Library Accomplishments and Future Directions. 1999.

[DMMP06] Allen H Dutoit, Raymond McCall, Ivan Mistrík, and Barbara Paech. *Rationale Management in Software Engineering*. Springer-Verlag, New York, 2006.

[DP02] Allen H Dutoit and Barbara Paech. Rationale-Based Use Case Specification. *Requirements Engineering*, 7(1):3–19, April 2002.

[EC75] Kapali P Eswaran and Donald D. Chamberlin. Functional specifications of a subsystem for data base integrity. In *1st International Conference on Very Large Data Bases*, pages 48–68, Framingham, Massachusetts, September 1975. ACM.

[FIC07] Syed Ahsan Fahmi, Ahmad Ibrahim, and Ho-Jin Choi. Enhancing Requirements Engineering Activities through the Use of Mobile Technology Devices and Tools. In *Future Generation Communication and Networking*, pages 578–581, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[Fre] Free Software Foundation. GNU Bison [online]. Available from: http://www.gnu.org/software/bison/.

[GF94] Orlena C Z Gotel and Anthony C W Finkelstein. An Analysis of the Requirements Traceability Problem. In *First International Conference on Requirements Engineering*, pages 94–101, 1994.

[GHJV96] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Longman, Amsterdam, 1st edition, 1996.

[HSH93] Debby Hindus, Chris Schmandt, and Chris Horner. Capturing, structuring, and representing ubiquitous audio. *ACM Transactions on Information Systems*, 11(4):376–400, 1993.

[Ins06] Institute of Electrical and Electronics Engineers. IEEE 1074-2006, 2006.

[Int07] International Organization for Standardization. *ISO/IEC 19502:2005, Information technology - Meta Object Facility (MOF)*. Distributed through American National Standards Institute (ANSI), August 2007.

[Int09] International Organization for Standardization. ISO 31000:2009 Risk management - Principles and guidelines, November 2009.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1st edition, February 1999.

[JT79] Randall W. Jensen and Charles C. Tonies. *Software Engineering*. Prentice Hall, May 1979.

[KFRC90] Robert E Kraut, Robert S Fish, Robert W Root, and Barbara L Chalfonte. Informal Communication in Organizations. *Human Reactions to Technology: The Claremont Symposium on Applied Social Psychology*, 1990.

[KJW59] James E Kelley Jr and Morgan R Walker. Critical-Path Planning and Scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173, New York, New York, USA, 1959. ACM Press.

[Kög08] Maximilian Kögel. Towards software configuration management for unified models. In *CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*. ACM, May 2008.

[Kög11] Maximilian Kögel. *Operation-based model evolution*. Verlag Dr. Hut, 2011.

[KR70] Werner Kunz and Horst W J Rittel. Issues as elements of information systems. Technical report, 1970.

[KSM84] Sara Kiesler, Jane A L Siegel, and Timothy W McGuire. Social psychological aspects of computer-mediated communication. *American Psychologist*, 39:1123–1134, October 1984.

[LEG+02] Dar-Shyang Lee, Berna Erol, Jamey Graham, Jonathan J. Hull, and Norihiko Murata. Portable meeting recorder. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 493–502, New York, NY, USA, 2002. ACM.

[LMB92] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc*. O'Reilly Media, 1992.

[MFM+95] Carl Mazza, John Fairclough, Brian Melton, Daniel De Pablo, and Adrian Scheffer. *Software Engineering Standards*. Prentice Hall, 1st international edition, January 1995.

[MR99] Robert Müller and Erhard Rahm. Rule-Based Dynamic Modification of Workflows in a Medical Domain. In *Proceedings of BTW99*, pages 429–448, Freiburg im Breisgau, 1999.

[MRCF59] Donald G Malcolm, John H Roseboom, Charles E Clark, and Willard Fazar. Application of a Technique for Research and Development Program Evaluation. *Operations Research*, 7(5):646–669, 1959.

[MYBM91] Allan MacLean, Richard M Young, Victoria M. E. Bellotti, and Thomas P Moran. Questions, options, and criteria: elements of design space analysis. *Human-Computer Interaction*, 6(3):201–250, September 1991.

[MYM89] Allan MacLean, Richard M Young, and Thomas P Moran. Design rationale: the argument behind the artifact. In *CHI '89: Proceedings of the SIGCHI conference on Human factors in computing systems: Wings for the mind*. ACM Request Permissions, March 1989.

[NHKN10] Michael Nagel, Jonas Helming, Maximilian Koegel, and Helmut Naughton. Audio Recording in Software Engineering. In *FlexiTools workshop at ICSE 2010, Cape Town*, 2010. Available from: http://wwwbruegge.in.tum.de/static/publications/pdf/214/FlexiTools.pdf.

[NT95] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company*. How Japanese Companies Create the Dynamics of Innovation. Oxford University Press, USA, May 1995.

[OA06] Jun Ogata and Futoshi Asano. Stream-Based Classification and Segmentation of Speech Events in Meeting Recordings. In *Multimedia Content Representation, Classification and Security*, pages 793–800. 2006.

[Obj11a] Object Management Group. BPMN 2.0 Specification. OMG, January 2011. Available from: http://www.omg.org/spec/BPMN/2.0/.

[Obj11b] Object Management Group. Meta Object Facility Specification [online]. August 2011. Available from: http://www.omg.org/spec/MOF/.

[PDN+07] Matthew Purver, John Dowding, John Niekrasz, Patrick Ehlen, Sharareh Noorbaloochi, and Stanley Peters. Detecting and summarizing action items in multiparty dialogue. In *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue*, 2007.

[PEN06] Matthew Purver, Patrick Ehlen, and John Niekrasz. Detecting action items in multi-party meetings: Annotation and initial experiments. *MLMI'06 Proceedings of the Third international conference on Machine Learning for Multimodal Interaction*, pages 200–211, 2006.

[RAG+01] Heather Richter, Gregory Abowd, Werner Geyer, Ludwin Fuchs, Shahrokh Daijavad, and Steven Poltrock. Integrating Meeting Capture within a Collaborative Team Environment. In *Ubicomp 2001: Ubiquitous Computing*, pages 123–138, 2001.

[RGLL06] Heather Richter, Robin Gandhi, Lei Liu, and Seok-Won Lee. Incorporating Multimedia Source Materials into a Traceability Framework. In *Proceedings of the First International Workshop on Multimedia Requirements Engineering*, page 7, 2006.

[Roy70] Winston W Royce. Managing the Development of Large Software Systems. In *IEEE WESCON*, 1970.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, October 2001.

[Sch96] Ken Schwaber. SCRUM Development Process. *OOPSLA'95 Business Object Design and Information Workshop*, pages 1–23, April 1996.

[Sch00] Michael E Schmidt. *Implementing the IEEE software engineering standards*. Sams, Indianapolis, 2000.

[Sch04] Ken Schwaber. *Agile project management with Scrum*. Microsoft Press, 2004.

[Sch06] Kurt Schneider. Rationale as a By-Product. In *Rationale Management in Software Engineering*, pages 91–109. Springer-Verlag, 2006.

[Sch11] Jennifer Schiller. *A Framework for Externalizing Information in Agile Meetings*. Dissertation, Technische Universität München, München, 2011.

[SCY05] Rainer Stiefelhagen, Xilin Chen, and Jie Yang. Capturing Interactions in Meetings with Omnidirectional Cameras. *International Journal of Distance Education Technologies*, 3(3):34–47, 2005.

[SO98] Shazia W. Sadiq and Maria E. Orlowska. Dynamic modification of workflows. Technical report, 1998.

[SS00] Nitin Sawhney and Chris Schmandt. Nomadic radio: speech and audio interaction for contextual messaging in nomadic environments. *ACM Transactions on Computer-Human Interaction*, 7(3):353–383, 2000.

[Sto75] Michael Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *Proceedings of the 1975 ACM SIGMOD international conference on Management of data - SIGMOD '75*, page 65, New York, New York, USA, 1975. ACM Press.

[THH03] Roger C. F. Tucker, Marianne Hickey, and Nick Haddock. Speech-as-data technologies for personal information devices. *Personal Ubiquitous Computing*, 7(1):22–29, 2003.

[TN86] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Harvard business review*, January 1986.

[WD04] Timo Wolf and Allen Dutoit. A Rationale-based Analysis Tool. In *Proceedings of the ISCA 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, pages 209–214. ISCA, July 2004.

[Wol07] Timo Wolf. *Rationale-based Unified Software Engineering Model*. Dissertation, Technische Universität München, 2007.

List of Figures

Listings

APPENDIX A

Full QQL Grammar

This appendix lists the complete LALR(1) grammar for QQL in EBNF form.

```
qql ::= ( transaction | validatordefinition | functiondefinition |
   variabledeclaration )+
transaction ::= 'transaction' statementblock
statementblock ::= '{' statement ( ';' statement )* '}'
statement ::= variabledeclaration | assignment | while | for | if |
   functioncall | return
assignment ::= variable '=' expression
expression ::= literal | variable | functioncall | '!' '(' expression
    ')' | '(' expression operator expression ')'
operator ::= equalop | booleanop | compareop
equalop ::= '==' | '!='
booleanop ::= '||' | '&&'
compareop ::= '<' | '<=' | '>' | '>='
variabledeclaration ::= vartype variable ( '=' expression )?
vartype ::= 'Bool' | 'String' | 'Numeric' | 'Date' | 'Quarc' | '
   Attribute' | 'AttrEnumItem' | 'QuarcRelation' | 'CommRelation' | '
   Segment' | 'List'
variable ::= identifier
if ::= 'if' '(' expression ')' statementblock ( 'else' statementblock
    )?
while ::= 'while' '(' expression ')' statementblock
for ::= 'for' vartype? variable 'in' expression statementblock
functiondefinition ::= 'function' vartype? functionname '(' (
   parameter ( ',' parameter )* )? ] ')' statementblock
functionname ::= identifier
parameter ::= ( 'Entity' | vartype) identifier
return ::= 'return' [ expression ]
functioncall ::= functionname '(' ( expression ( ',' expression )* )?
    ')'
validatordefinition ::= 'validator' validatorname statementblock
```

```
validatorname ::= identifier
```

Listing A.1: LALR(1) grammar for QQL

The following *bison* [Fre] file generates a lexer/parser for QQL. Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. It accepts the same input syntax as yacc [LMB92] with the addition of C-style string literals, where yacc only understands single-character token literals.

```
%token IDENTIFIER
%token LITERAL

%%

qql : topblocklist;

topblocklist : topblock | topblocklist topblock
topblock : statementblock | validatordefinition | functiondefinition
   | variabledeclaration;
statementblock : "{" statementlist "}";
statementlist : statement | statementlist ";" statement;
statement : variabledeclaration | assignment | while | for | if |
   functioncall | return;
assignment : variable "=" expression;
expression : LITERAL | variable | functioncall | "!" "(" expression
   ")" | "(" expression operator expression ")";
operator : equalop | booleanop | compareop;
equalop : "==" | "!=";
booleanop : "||" | "&&";
compareop : "<" | "<=" | ">" | ">=";
variabledeclaration : vartype variable declassign;
declassign : /* empty */ | "=" expression
vartype : "Bool" | "String" | "Numeric" | "Date" | "Quarc" | "
   Attribute" | "AttrEnumItem" | "QuarcRelation" | "CommRelation" | "
   Segment" | "List" ;
variable : identifier;
identifier : IDENTIFIER;
if : ifthen | ifthenelse;
ifthen: "if" "(" expression ")" statementblock
ifthenelse : ifthen "else" statementblock
while : "while" "(" expression ")" statementblock;
for : "for" forvar "in" expression statementblock;
forvar : variable | vartype variable
functiondefinition : "function" functype functionname "("
   funcparameterdeclist ")" statementblock;
functype : /* empty */ | vartype
functionname : identifier;
funcparameterdeclist : /* empty */ | parameterdeclist
```

```
parameterdeclist : parameterdec | parameterdeclist "," parameterdec
parameterdec : partype identifier;
partype : "Entity" | vartype
return : "return" "(" returnvalue ")";
returnvalue : /* empty */ | expression;
functioncall : functionname "(" funcparameterlist ")";
funcparameterlist : /* empty */ | parameterlist
parameterlist : parameter | parameterlist "," parameter
parameter : expression;
validatordefinition : "validator" validatorname statementblock;
validatorname : identifier;

%%
```

APPENDIX B

Default Quarc Repository Validators

The default QUARC repository model (see subsection 2.6.3) defines the three quarc relations startDependsOn, closeDependsOn and alternativeCloseDependsOn that express different kinds of dependencies between quarcs (see subsection 2.3.3). To enforce these dependencies as structural integrity constraints on the quarc repository level, we need to implement validators. The following three listings show these validators implemented in QQL.

```
validator checkStartDependsOn {
   for Quarc x in changedQuarcs() {
    AttrEnumItem status_new;
    status_new = getAttributeValueEnumForQuarc(x, #status);
    AttrEnumItem status_old;
    status_old = getAttributeValueEnumForQuarc(x, #status);

    if (status_new != status_old) {
     // Is the quarc starting?
     if (status_new == #inProgress) {
      // Check outgoing relations
      List dep;
      dep = quarcsRelatedFromQuarc(x, #startDependsOn);
      for Quarc y in dep {
       AttrEnumItem status_rel;
       status_rel = getAttributeValueEnumForQuarc(y, #status);
       assert((status_rel == #closed) || (status_rel == #dormant));
      }
     }

     // Is the quarc reopening?
     if ((status_old == #closed) && (status_new != #dormant)) {
      // Check incoming relations
      List dep;
      dep = quarcsRelatedToQuarc(x, #startDependsOn);
      for Quarc y in dep {
       AttrEnumItem status_rel;
```

```
          status_rel = getAttributeValueEnumForQuarc(y, #status);
          assert((status_rel == #open) || (status_rel == #dormant));
        }
      }
    }
  }
}
```

Listing B.1: Validator for the default relation startDependsOn in QQL

```
validator checkCloseDependsOn {
   for Quarc x in changedQuarcs() {
    AttrEnumItem status_new;
    status_new = getAttributeValueEnumForQuarc(x, #status);
    AttrEnumItem status_old;
    status_old = getAttributeValueEnumForQuarc(x, #status);

    if (status_new != status_old) {
      // Is the quarc closing?
      if (status_new == #closed) {
        // Check outgoing relations
        List dep;
        dep = quarcsRelatedFromQuarc(x, #closeDependsOn);
        for Quarc y in dep {
          AttrEnumItem status_rel;
          status_rel = getAttributeValueEnumForQuarc(y, #status);
          assert((status_rel == #closed) || (status_rel == #dormant));
        }
      }

      // Is the quarc reopening?
      if ((status_old == #closed) && (status_new != #dormant)) {
        // Check incoming relations
        List dep;
        dep = quarcsRelatedToQuarc(x, #closeDependsOn);
        for Quarc y in dep {
          AttrEnumItem status_rel;
          status_rel = getAttributeValueEnumForQuarc(y, #status);
          assert(status_rel != #closed);
        }
      }
    }
   }
  }
}
```

Listing B.2: Validator for the default relation closeDependsOn in QQL

```
validator checkAlternativeCloseDependsOn {
   for Quarc x in changedQuarcs() {
     AttrEnumItem status_new;
     status_new = getAttributeValueEnumForQuarc(x, #status);
     AttrEnumItem status_old;
     status_old = getAttributeValueEnumForQuarc(x, #status);

     if (status_new != status_old) {
       // Is the quarc closing?
       if (status_new == #closed) {
         // Check outgoing relations
         List dep;
         dep = quarcsRelatedFromQuarc(x, #alternativeCloseDependsOn);
         BOOL found = FALSE;
         for Quarc y in dep {
           AttrEnumItem status_rel;
           status_rel = getAttributeValueEnumForQuarc(y, #status);
           if (status_rel == #closed) {
             found = TRUE;
           }
           assert(found);
         }
       }

       // Is the quarc reopening?
       if ((status_old == #closed) && (status_new != #dormant)) {
         // Check incoming relations
         List dep;
         dep = quarcsRelatedToQuarc(x, #alternativeCloseDependsOn);
         for Quarc y in dep {
           // Check outgoing relations of y
           List dep;
           dep = quarcsRelatedFromQuarc(y, #alternativeCloseDependsOn);
           BOOL found = FALSE;
           for Quarc z in dep {
             AttrEnumItem status_rel;
             status_rel = getAttributeValueEnumForQuarc(z, #status);
             if (status_rel == #closed) {
               found = TRUE;
             }
             assert(found);
           }
         }
       }
     }
   }
}
```

Listing B.3: Validator for the default relation alternativeCloseDependsOn in QQL

APPENDIX C

Case Study on Audio Recordings in Project Management

In this appendix, we present a small qualitative case study that investigates the practicability of including voice recordings in a project management system.

C.1 Environment

C.1.1 UNICASE

One of the long-term research projects at the Chair for Applied Software Engineering at TUM is the CASE platform *UNICASE* [BCH07], formerly named *Sysiphus* [BDW06]. UNICASE evolved from the earlier projects *ReQUEST* [DP02] and *RAT* [WD04]. UNICASE provides an initial set of model elements from two domains, the system model and the project model. Model elements from the system model describe the system under construction, such as functional requirements, use cases, components or UML artifacts. Model elements from the project model describe tasks, bug reports, the organizational structure, iterations, and meetings. The models are stored and versioned on a server comparable to the Subversion software configuration management system, but customized for models. UNICASE is based on the Eclipse platform including EMF and GMF. EMFStore is the versioned EMF model storage server/client, and ECP (EMF Client Platform) is an application generator that creates applications from EMF models.

C.1.2 UniRec

The goal of our study was to investigate if it is feasible, and useful, to record informal conversations and include them in UNICASE. The functional requirements for our study were to allow developers to record audio in a mobile environment, upload these recordings to a UNICASE project, and to link them to UNICASE project model elements. In order to fulfill these requirements, we built an iOS app named *UniRec* (*UNI*CASE *Rec*order) that was capable

of recording audio, and an intermediate infrastructure (UniRec Proxy Server and WebDAV-enabled file repository) to facilitate communication with the UNICASE server. Figure C.1 shows the software architecture. We describe the proxy server in more detail later in this section.

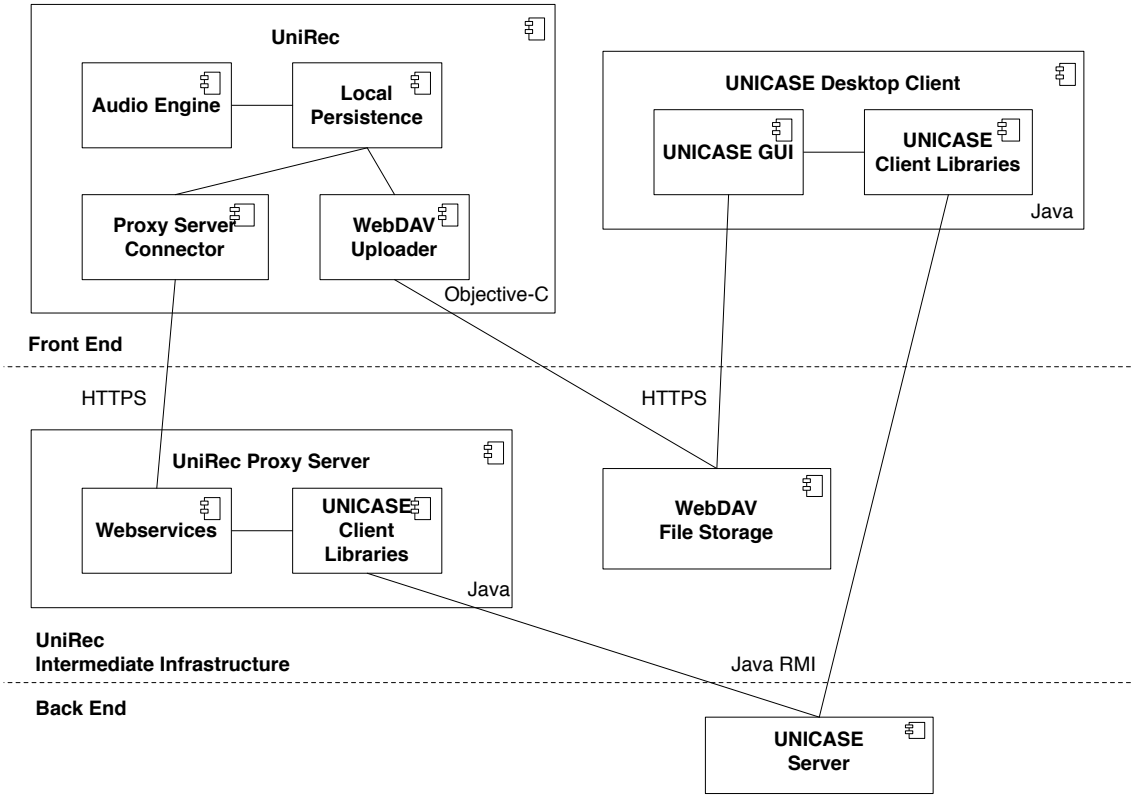


Figure C.1: UniRec architecture

UniRec offers dictaphone-style recording controls (see figure C.2) (using the audio engine from the Dictamus app). After finishing a recording, the user uploads it to the WebDAV file repository.

There was only preliminary support for uploading files directly to a UNICASE server. We therefore decided to set up a separate WebDAV file repository to host the audio files, and provide links to these audio files from UNICASE in the form of URL attachments to the respective model elements.

To attach an audio recording to a model element, UniRec enables the user to choose a UNICASE project, and navigate that project's project model to locate the target model element. For navigation in the project model, we used a simple hierarchical iOS navigation controller approach. A project model in UNICASE is organized as a tree of model elements, with the model element representing the project itself as the root node of the tree. This structure is well suited for browsing in navigation controller hierarchies. UniRec allows the user to browse the entire UNICASE project model tree to choose the model element to which the audio recording should

Figure C.2: UniRec recording controls

be attached.

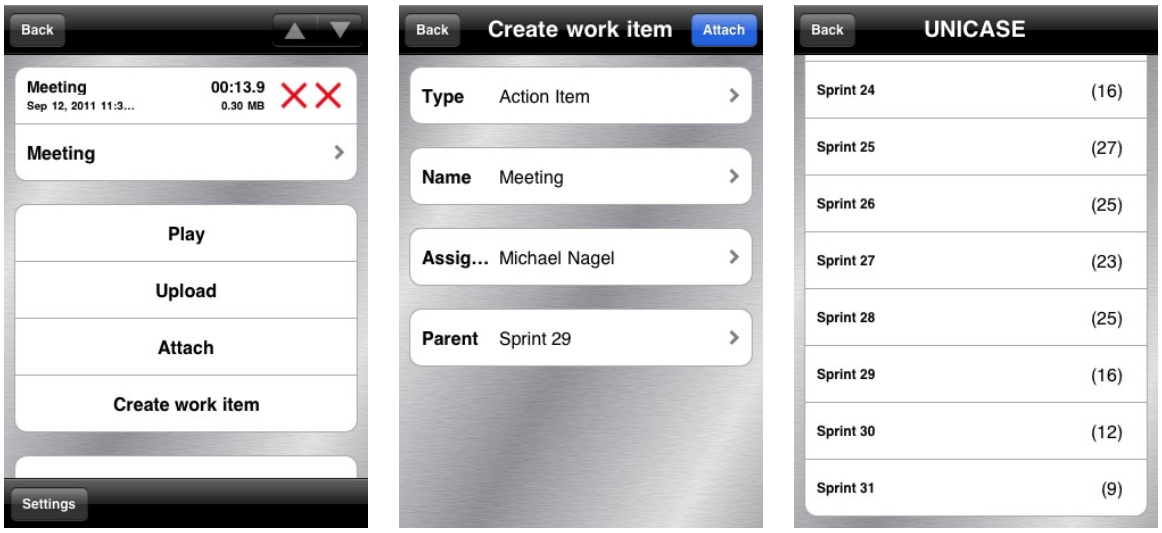
Creating UNICASE model elements for linking audio recordings requires specialized code in the proxy server, and a specialized user interface in UniRec for each supported model element type. UNICASE has over 40 different model element types, so we limited the scope of this capability to UNICASE *work items*.

A UNICASE work item is either an *Issue*, a *Bug Report* or an *Action Item*. These model element types are closely related, which reduced the implementation effort on both sides considerably. Figure C.3 shows the user interfaces for the three most important steps to create a new work item.

The left screen shows the *Details* view used for managing a single audio recording. In order to create a work item, the user first uploads the recording to the WebDAV repository. Next, he uses the *Create Work Item* button to move to the respective screen, shown in the middle screen. After entering a description and choosing an assignee, he chooses the *Work Package* in which he wants to create the work item.

In a typical UNICASE project, the parent of a work item is a work package. Therefore, in work item mode, the UNICASE project navigator filters out all model elements except composites (generic parent nodes) and work packages (shown in the right screen of figure C.3).

Direct communication between UniRec and the UNICASE would of course have been preferred over an additional proxy server. The existing UNICASE client libraries were only available in Java language, but Java is not supported on iOS. Furthermore, the UNICASE server didn't offer any suitable interfaces for communication that bypasses the client libraries at all. Communication between a UNICASE server and its clients was exclusively performed over Java RMI (Remote Method Invocation), which is not available to Objective-C, the native iOS



(a) Recording details **(b)** Create Work Item **(c)** Work Package Navigator

Figure C.3: Creating a work item in UniRec

language. An architectural decision that was making this particular problem even worse was that the UNICASE client libraries were not just communication wrappers, they also implemented client-side persistence and automatic change tracking (see [Anob]). The client-side libraries may even use EMF capabilities, such as for serialization/deserialization.

A direct communication from a native iOS app to a UNICASE server would involve Objective-C implementation of several components from scratch, such as Java's RMI framework, UNICASE's local persistence and change tracking mechanisms, and parts of the EMF framework. We therefore decided to introduce an intermediate server, the UniRec Proxy Server. The proxy server was implemented in Java, so it could use the existing UNICASE client libraries, and exposed a set of specialized web services to communicate with UniRec in XML over standard HTTPS connections, which are natively supported by the iOS platform. We set up the following web services:

- Get the list of UNICASE projects available on the proxy server
- Get all children of a specific model element (or of the root node) in a project, optionally filtered for composites and work packages. Return name, id, model element type and number of own children for each item.
- Get a list of all users in a project
- Attach an audio recording to a specific model element
- Create a new work item with the given type, description, and assignee under the given parent, and attach an audio recording

Due to the brevity of that list and the lack of support on the iOS side, we chose not to use a standard web service framework, but instead to code these services by hand. Using a standard

framework might have shortened the implementation time on the proxy server side, but would at the same time have increased the effort on the iOS side considerably.

In the standard desktop UNICASE client, the linked audio recording attachments were visible as standard URL attachments, like a website address attached to a model element. Clicking such an attachment opened the audio recording in a new browser window and played it. An inline solution, i.e. a "Play" button in the UNICASE client, would have been more user-friendly, but would have required significant additional development time due to the many levels of abstraction and indirection in UNICASE. Many of these are dictated by the extensive use of dynamic techniques such as Java Reflection, employed to enable UNICASE to adapt to modifiable EMF (Ecore) base models, and the modular nature of UNICASE and the underlying Eclipse RCP platform. Furthermore, UNICASE was a moving target due to the active development at the time of the case study, with frequent API changes due to refactorings. Therefore, UniRec used URL attachments and browser playback for the case study.

C.2 Case Study

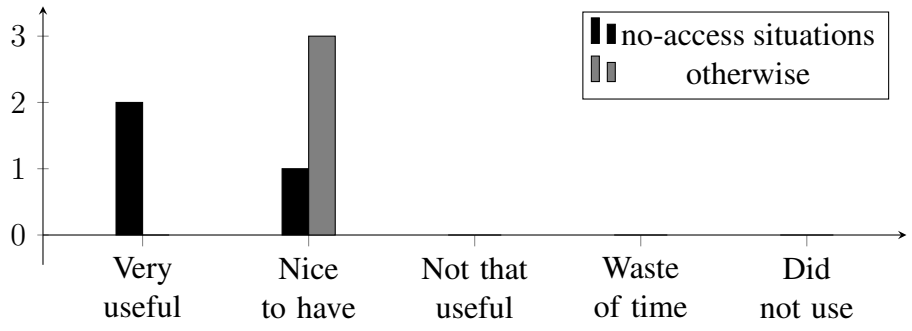
The purpose of our case study was to investigate the use of audio recordings during the development of UNICASE. The participants were 20 developers of the UNICASE developer team. We refer to the participants in this study as *Producers* – two project lead and one developer – and *Recipients*. The Producers created audio recordings using iPhone and iPod touch devices. They recorded planned and informal one-to-one meetings, and created and assigned work items based on audio recordings. In the QUARC model, these recordings can be modeled as communication artifacts and communication segments (see section 2.4). The UNICASE work items to which the audio recordings were attached can be represented as quarcs. The Recipients were the assignees of the work items created by the Producers.

We examined both, the creator of the audio recording (Producer) and the consumer (Recipient), because our hypothesis was that Producers would rather speak than type, and Recipients would rather read than listen. By investigating both sides, we planned to get an impression of the overall effect on a project. We only had a limited response of three answers for each of the both sides, so we can present our results only as anecdotal evidence.

C.2.1 Producer Side

We distributed a Producer survey to the core developer team. All three submitted answers.

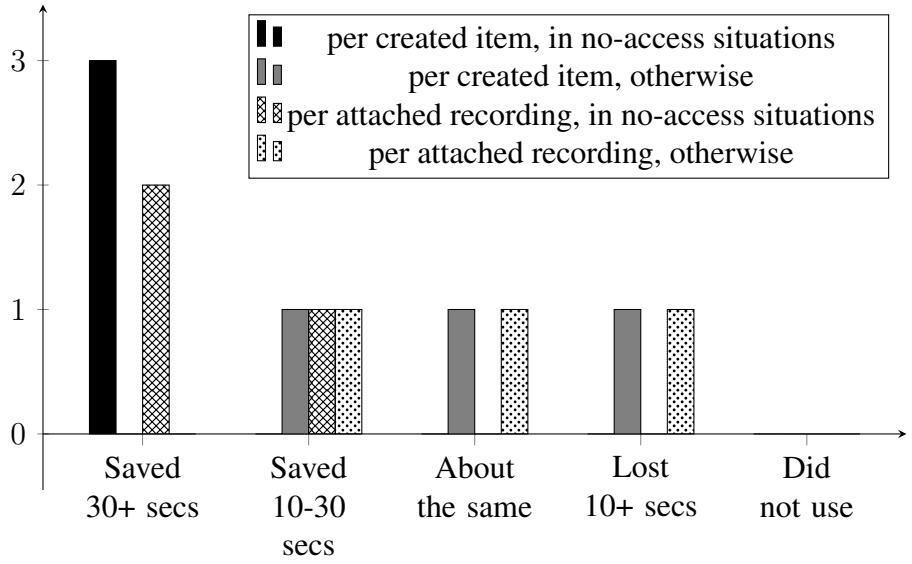
The three Producers considered the capability to record audio instead of typing text as useful, especially in *no-access situations*, that means mobile situations where they had no access to a desktop UNICASE client.



UniRec allowed the Producers to create new UNICASE work items for recordings, and to attach recordings to existing UNICASE model elements. The Producers used both capabilities, with a preference towards action items, both for newly created and existing model elements. Bug reports were the second most created/attached model element type.

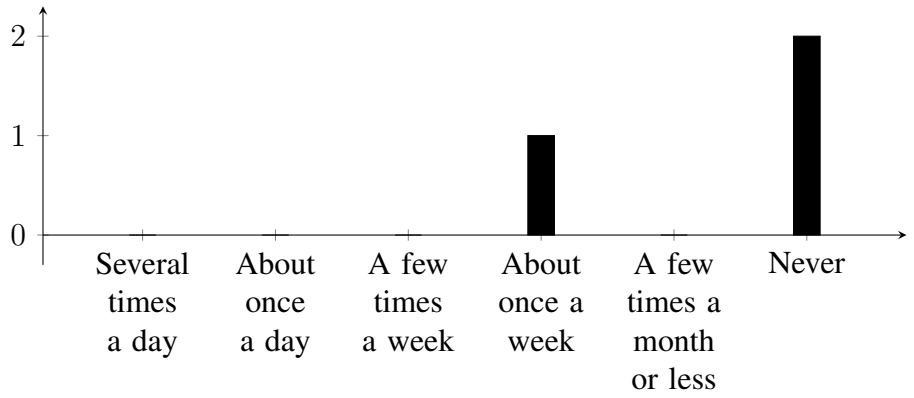
All three Producers used UniRec to perform their usual tasks, in particular the creation of work items. For these tasks, they regarded the functionality offered by UniRec as time-saving compared to traditional entry methods in no-access situations. When a traditional desktop client was available, the answers were distributed evenly around the "took about the same time" mark, so at least they didn't lose time in these situations.

We conclude that Producers save time overall when using audio recordings.



We also asked the Producers to use UniRec to record informal communication and meetings in order to capture rationale. One Producer was especially enthusiastic about this in the pre-study interview, because he envisioned recording one-to-one student meetings for later revision by both sides. He stated that in this kind of meeting, participants usually made no explicit notes of arguments or even action items, because both were sure to remember them later anyway. Often, though, both participants happened to forget them. Our hypothesis was that an audio

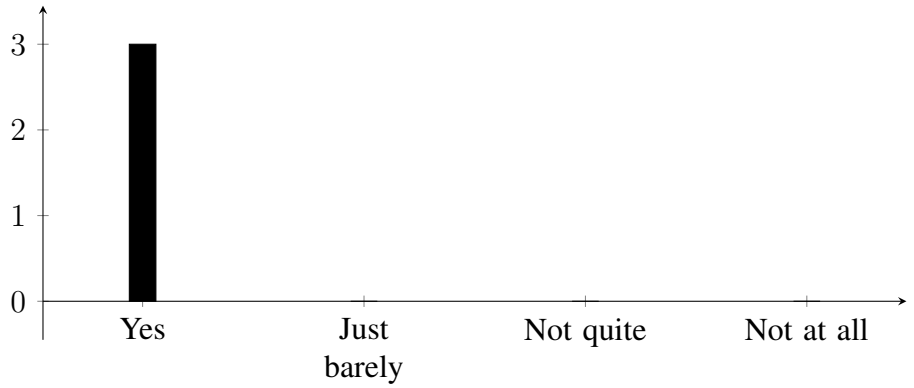
recording of the informal meeting would address this problem.



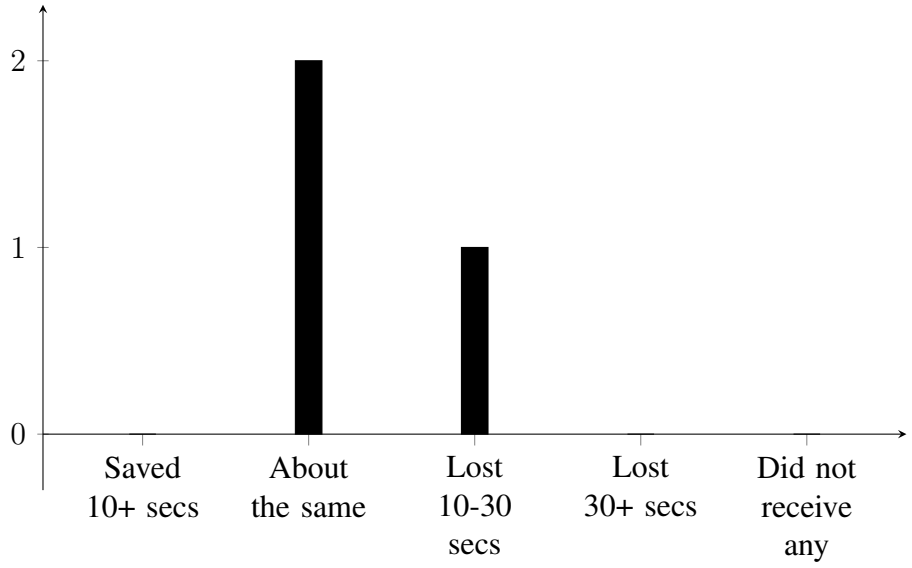
It turned out that only one of the Producers recorded any informal meetings at all, and did so only about once a week. This is surprising because it wasn't even the Producer who was enthusiastic about this feature at first. In fact, he didn't record any informal meetings at all. In a retrospective interview, he found it too cumbersome to retrieve his device, plug in the external microphone, and wait for UniRec to launch before engaging in a discussion. Most of these usability issues should be addressable with always-available, always-on devices such as the users' own smartphones. There seems to be a psychological factor at work as well, because the time lost by these usability issues is small compared to the length of such a meeting (approx. 20 seconds vs. 20-60 minutes).

C.2.2 Recipient Side

We distributed a second survey to all developers that were active in the UNICASE project during the study. The Recipients considered working with audio attachments generally as acceptable. This contradicts our initial hypothesis that Recipients prefer to read rather than listen.



The Recipients didn't lose much time when processing audio-based work items compared to text-based work items.



The Recipients agreed that the user interface for audio attachments in UNICASE had quite some potential for improvement. They requested an audio player integrated in the desktop UNICASE client instead of playing audio in a separate browser. They also asked for better discoverability for audio recording attachments. The UNICASE user interface was not specifically restructured for the case study to show audio recordings close to description and name of the model element, instead it was listed among the attachments of the element at a peripheral position. A QUARC-based tool would treat all communication segment types as equal, so audio recordings would be just as visible as textual attachments.

APPENDIX D

Quarc Repository Models

Throughout this dissertation, we present several quarc repository models. In this appendix, we collect all these models.

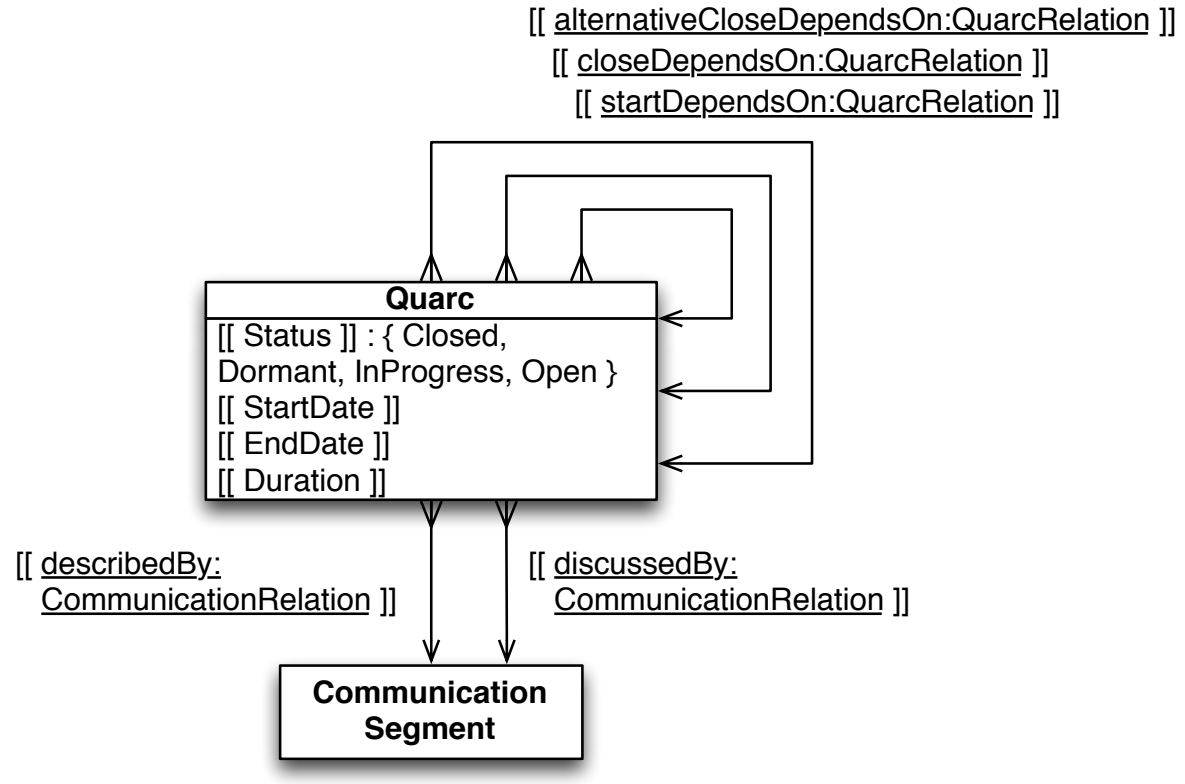


Figure D.1: The Q2 default quarc repository

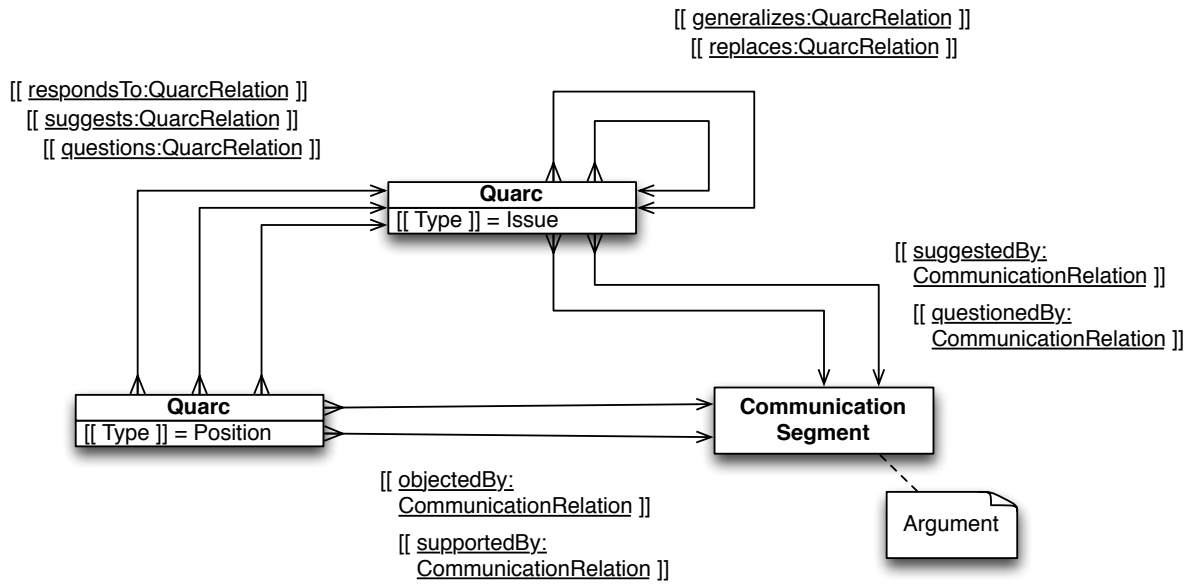


Figure D.2: IBIS represented as a QUARC repository model

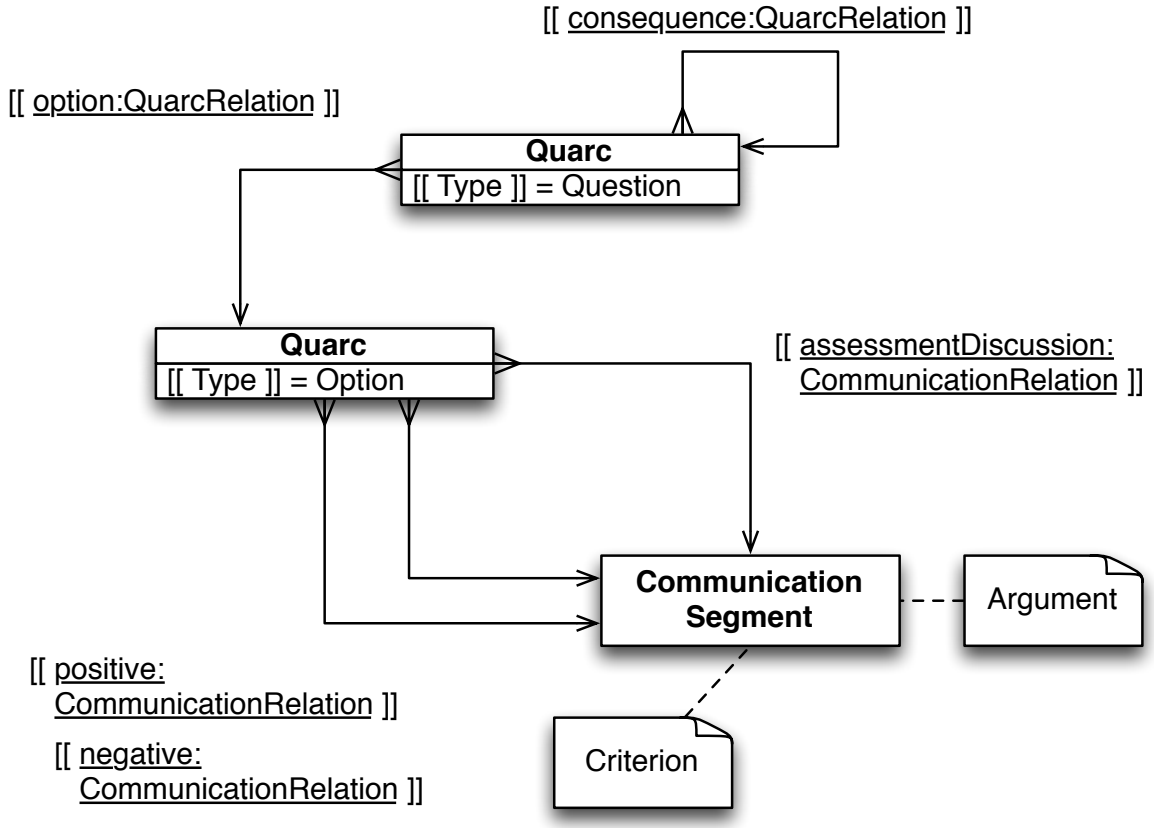


Figure D.3: The QOC entity model represented as a quarc repository model

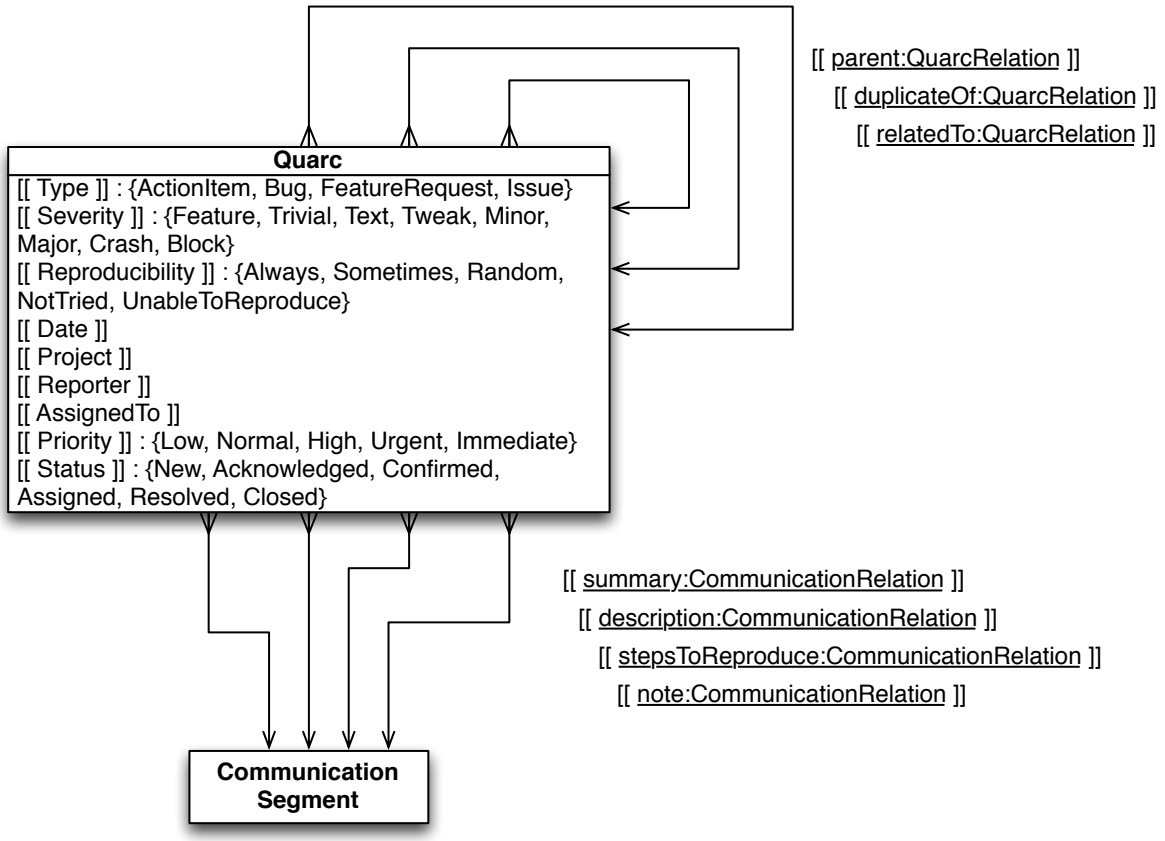


Figure D.4: The Mantis entity model represented as a QUARC repository model

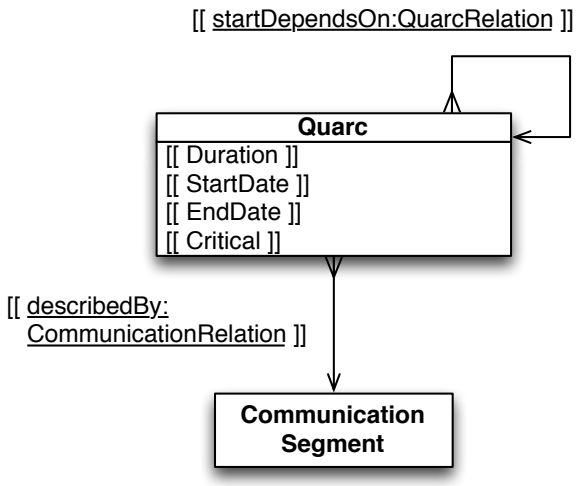


Figure D.5: PERT/CPM as a QUARC repository model

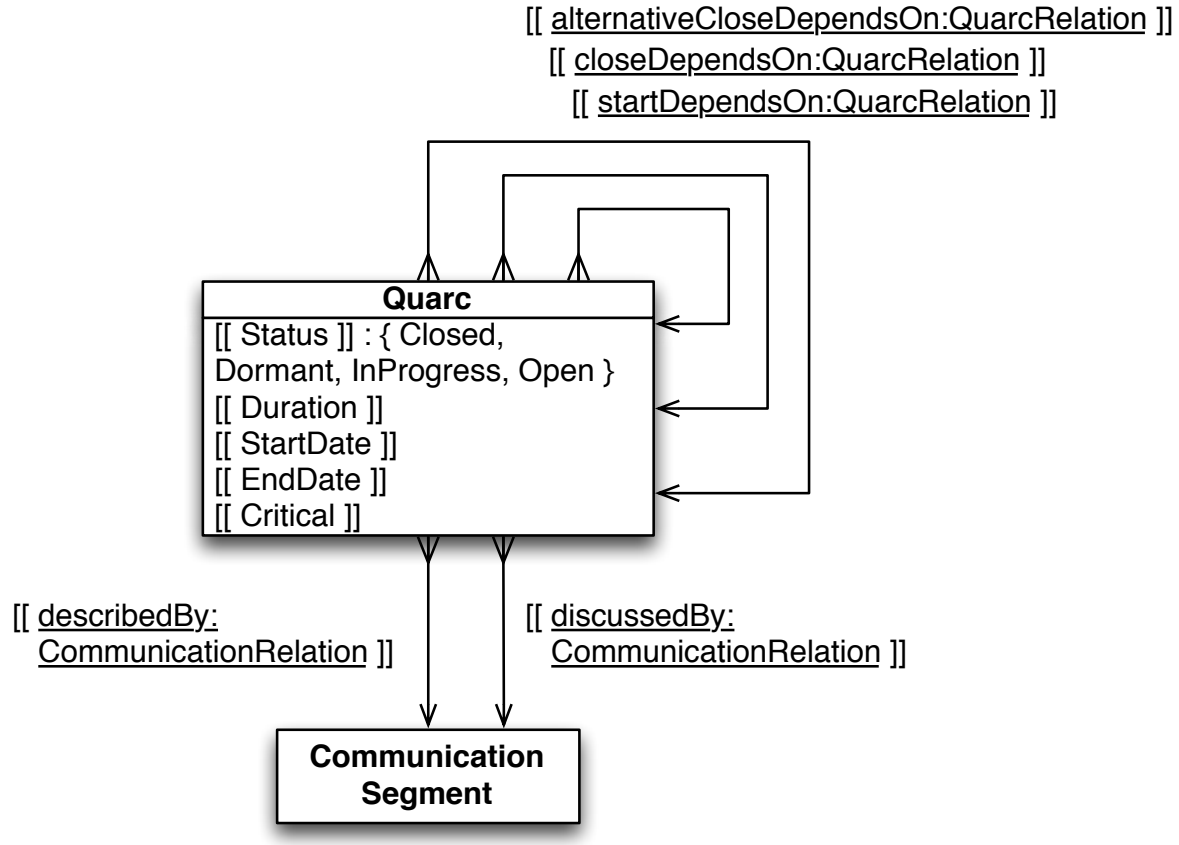


Figure D.6: Hybrid task/issue/risk management quarc repository model