Electronic version of an article published as **IJSEKE, Vol. 16, Issue 6, 2006, pp. 917-950** [doi:10.1142/S021819400600304X]

International Journal of Software Engineering
and Knowledge Engineering
Vol. 16, No. 6 (2006) 1–34

ICRAD: AN INTEGRATED PROCESS FOR THE SOLUTION OF REQUIREMENTS CONFLICTS AND ARCHITECTURAL DESIGN*

ANDREA HERRMANN† and BARBARA PAECH‡

Software Engineering Group, Faculty of Mathematics and Computer Science,
University of Heidelberg, 69120 Heidelberg, Germany
†*herrmann@informatik.uni-heidelberg.de*
‡*paech@informatik.uni-heidelberg.de*
http://www-swe.informatik.uni-heidelberg.de/

DAMIAN PLAZA

Institut für Medizinische Biometrie und Informatik,
University of Heidelberg, 69120 Heidelberg, Germany
http://www.biometrie.uni-heidelberg.de
Interdisziplinäres Uveitiszentrum Heidelberg, 69120 Heidelberg, Germany
http://www.uveitiscenter.de

In order to solve requirements conflicts when developing or enhancing an IT system, it is essential to understand its architecture. Frequently, the costs for the realization of certain requirements are a decision criterion. Requirements negotiation and architectural design must be treated together, as conflicts cannot be solved before the architecture has been designed. So far, no integrated process exists which clearly defines input and output among these activities, and which takes into account a variety of different types of dependencies among requirements and between requirements and architecture.

In this paper, we develop a detailed iterative process named ICRAD (Integrated Conflict Resolution and Architectural Design). ICRAD integrates requirements negotiation and architectural design and takes into account nine types of dependencies. We define three types of requirements conflicts. We also present a case study.

Keywords: Requirements engineering; requirements conflicts; requirements negotiation; requirements decisions; architectural design; design decisions.

1. Introduction

The route from the requirements of an IT system to its architectural design is called the route from *problem space* (also called *requirements space*) to *solution space*. Along this way, conflicts among requirements must be solved and architectural

*This work is the result of the research project SIKOSA, which is funded by the Ministry for Science, Research and Art of Baden-Württemberg, Germany (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg).

alternatives must be compared with each other. This signifies a complex decision-making process.

We define the interface between requirements and solution space like this: Requirements are wishes. They might contradict each other or not be realizable. They are specified relatively independent of each other. The architectural design describes a realizable solution. Whether it satisfies all of the requirements has to be evaluated. According to this definition, even architectural requirements derived during the requirements elicitation are requirements, although they describe desirable parts or properties of an architectural design. Still they are wishes which might be unrealistic or contradicting. Seen from the perspective of the requirements engineer, architectural design is also a negotiation activity, which detects and solves feasibility conflicts among requirements.

The solution of requirements conflicts and architectural design depend on each other more strongly than is usually considered in the literature. Only a few requirements conflicts can be solved within the requirements space. The architectural design provides information about the feasibility and the costs of requirements, which should be put into the negotiation of requirements conflicts as important decision criteria. Therefore, it is neither sufficient nor possible to solve all requirements conflicts before the architectural design is set up.

We developed a systematic, iterative process, where requirements conflicts are translated into architectural decisions, and the decision-making in the solution space also has to solve the requirements conflicts. When making architectural decisions, several types of interdependencies among requirements have to be taken into account. Architectural decisions lead to new, changed or improved requirements, which then are the basis for a more detailed design. Following these principles, the activities, methods and concepts of many other authors are integrated.

Our work is not focused on architectural design *per se*, but on the solution of requirements conflicts based on architectural knowledge. Thus, we do not discuss different architectural styles, patterns, solutions or methods in designing IT architecture. We leave this to architecture specialists. We describe a process which systematically allows the use of architecture design knowledge in decision-making. It can be used in the development of a system from scratch or in the enhancement of a system.

Section 2 presents related work and Sec. 3 deals with the form of requirements we assume when they are put into our process. In Sec. 4, we identify various types of requirements conflicts, and in Sec. 5 we name the steps, activities, and methods of our process. Section 6 illustrates this process with a case study. Section 7 summarizes our work and contains the conclusions. Finally Sec. 8 sketches our plans for future work.

2. Related Work

We were looking for a systematic process which integrates the solution of requirements conflicts and architectural design, and solves requirements conflicts where necessary within the solution space. The process should also take into account several other types of requirements dependencies. We analyzed many approaches but found none which did use architectural design knowledge for solving requirements conflicts as comprehensively as we thought it necessary.

There is one work we want to mention because it has a similar goal like ours. Poort and de With [1] also develop a process for the detection and solution of conflicts and for high-level architectural design. Their process comprises similar steps like ours, but follows a different strategy: Requirements conflicts are solved in the problem space by splitting requirements groups. The authors justify their process by their daily experience and do not derive it systematically. We also want to mention that they do not discuss their criteria for prioritization and decision-making. Therefore it is not clear whether they use architectural knowledge for the solution of conflicts. We suspect that the authors, as it is usual in practical work, have integrated design and requirements engineering without being aware of it and that during the solution of requirements conflicts, architectural knowledge is used implicitly, as it becomes necessary. Therefore, their work does not contradict ours, but the integration of both aspects is not as radical as we wanted it to be.

Therefore, we derived a detailed process which explicitly uses architectural knowledge for the solution of requirements conflicts and which takes into account several other dependencies which are often neglected. In a first step, we identified a set of standard activities. To do so, our three main sources finally were: Sommerville [2] concerning validation of requirements, Robinson, Pawlowski, and Volkov [3] for conflict solution, and Bruegge and Dutoit [4] describing architectural design. Some other architectural design processes [5–14] and architectural analysis works [15–21] served as a source for activities and concepts of our process. We would like to discuss why one well-known example did not serve our purpose. ATAM (Architecture Tradeoff Analysis Method) [19] focuses on the decision-making during architectural design. It identifies decisions to be taken, evaluates the consequences of a decision, e.g. by a risk analysis, but: The prioritization of requirements is done by voting. No clear decision criteria are defined. It is not described how the initially proposed architectural design is built. Requirements conflicts are not explicitly solved, and only "functional dependencies" among requirements (corresponding to our feature bundle, see Sec. 5) are considered.

In other sources, very often the initial architectural outline is derived from the functional requirements, while the satisfaction of the non-functional requirements is used as a decision criterion for evaluating alternatives and incrementally improving the architecture [5, 8, 9]. Instead, we treat functional and non-functional requirements equally.

At the beginning of this section, we mentioned publications which describe processes and methods that cover more than one of our activities. The rest of this section discusses alternative methods for single activities of the process. The criterion for choosing a method was that its input fits the output of former activities and its output to the following activities; the order of the activities depended on where these inputs and outputs are produced and needed. The relationships among the activities are presented in Sec. 5. Here, alternatives from the literature are discussed.

2.1. *Requirements review and detection of requirements contradictions*

The criteria for a requirements review are defined by the Standard IEEE Std. 830-1998 [22].

Many authors advise that we concentrate on the **core requirements** only. Among them are Ruhe, Eberlein and Pfahl [23], who call them "mandatory requirements".

Zave and Jackson [24], treating consistency checking among different specifications, write: "We believe that the most practical consistency checking must be formulated at the same conceptual level as the specification languages used, and that algorithms for consistency checking will be specialized for particular languages and styles of decomposition." This reflects our own experience: For the **detection of requirements inconsistencies and contradictions** we could not use the detailed rules proposed for conflict detection by van Lamsweerde *et al.* [25]. They are only applicable to the formal and goal-oriented KAOS notation which they were designed for. Robinson, Pawlowski and Volkov [3] name the following methods for detection of requirements dependencies: classification-based, patterns-based, AI planning, scenario analysis, formal methods, runtime monitoring. We chose a classification based approach to detect conflicts by bundling requirements (also called separation of concerns or viewpoints [26–28]) and looking for the inconsistency types defined by van Lamsweerde *et al.* [25] and Grünbacher *et al.* [29], being generally more applicable to different forms of requirements. Clustering requirements for documenting dependencies is common practice, as reported by Dahlstedt and Persson [30] from an industry survey: "the requirements were clustered, usually with respect to which requirement that should be implemented together" (comparable to our "feature bundle", see Sec. 5). Based on a literature survey, the same authors [30] identify two main types of requirements interdependencies: structural dependencies (requires, explains, similar to, conflicts with, influences), and cost/value interdependencies (i.e. one requirement increases or decreases the value or cost of another requirement). These dependencies are considered in our work in different steps: by requirements bundles, conflicts and cost and benefit estimations based on a reference system.

2.2. *Solution of requirements conflicts within requirements space*

If inconsistent and contradicting requirements (definition in Sec. 4) are exclusive, then a clear decision has to be taken. In other cases, we refer to the conflict ("divergence") resolution strategies for partially conflicting requirements defined by van Lamsweerde *et al.* [25]. To solve requirements inconsistencies we can also refer to Robinson *et al.* [3] who name six strategies: relaxation (generalization or value-range extension), refinement specialization, compromise, restructuring (altering assumptions, reinforcement of a precondition to be satisfied, replanning), postponement, abandonment. These cover those defined by van Lamsweerde *et al.* [25]. As for solving conflicts among stakeholders and their inconsistent goals and requirements, the WinWin method is state of the art [29, 31, 32], and there are further works based on WinWin [33, 34]. Robinson *et al.* [3] propose the prioritization of decisions during the solution of requirements conflicts.

The idea of characterizing conflicts by their degree of conflict stems from Yen and Tiao [35]. They give a mathematical definition for this degree, but we prefer to simply estimate them.

2.3. *Identification of logical components*

Logical components bundle requirements according to architectural criteria, thus reducing the complexity of the requirements and also preparing the architectural design.

To identify logical components, the CBSP (Component-Bus-System and Properties) approach of Egyed and Gruenbacher [11, 36] could be used for classifying requirements according to the six CBSP dimensions. However, the method turned out to be highly recursive: Before attributing a requirement to a (logical) component, the logical components must be known. The CBSP output in its totality is not needed for the following steps of our process, and CBSP led to the same logical components as our approach.

2.4. *Architectural design*

As we said in the introduction, we did not want to compare architecture design methods, i.e. to discuss which parts an architectural design must have and in which order they must be derived. Our process can be combined with any design method. We based our work on Bruegge and Dutoit [4] which is consistent with the 4+1 view model of software architecture of Kruchten [37] (see their comparison in Sec. 5).

Mapping between requirements space and solution space: To do so, one can use a matrix [38] (our choice), graphical presentation [39, 40], or a special notation [41].

Evaluation of an architectural design: Alternatively to criteria benefit, cost, risk and complexity, one can measure the satisfaction of goals [42], or the satisfaction of high-level quality attributes like "performance" (see CBAM [43, 44]), or the weighted sum of how many scenarios are supported (see SAAM [45, 46]). However, the authors do not discuss their criteria for prioritization of these quality attributes and scenarios in detail.

The **detection/prediction of requirements conflicts (feasibility conflicts) in a technical solution** demands architectural experience. Often observed conflicts of non-functional requirements, i.e. experience like "security and computational efficiency, often conflict", is gathered by Egyed and Gruenbacher [47, 48], and also by Sutcliffe and Minocha [49].

The **trade-off between alternatives** in general and their documentation has been treated by a variety of researchers in the field of rationale. An overview over IBIS, PHI (Procedural Hierarchy of Issues), QOC (Questions, Options and Criteria), and DRL (Decision Representation Language) is given by Dutoit *et al.* [50]. Decisions can also be supported and documented by softgoal graphs [51, 52]. These methods allow arbitrary decision criteria. But as we managed to restrict the set of decision criteria on the benefit and (different types of) cost of requirements, we prefer our own documentation of negotiations in the template which is shown by Table 3. It allows a more comparable documentation.

Requirements conflict with other requirements, but also with project constraints like budget. The budget trade-off is a special case which can be treated in many ways. Ruhe, Eberlein and Pfahl [23] do so by maximizing the system value respecting the constraints of fixed effort, duration and quality by stepwise relaxation. (Remark: For them, technical feasibility is no criterion and cost is expected to be constant for each requirement, therefore architectural design needs not be considered explicitly.) The SQUARE project maximizes system value at a fixed project budget [53].

Review of Design and Identification of New Architectural Alternatives and Open Conflicts: Detailed questions for a design review can be found on page 281f of Bruegge and Dutoit [4].

3. Form of the Input Requirements

We assume that before our process starts, a requirements elicitation has been performed. The resulting requirements contain functional, non-functional, and architectural requirements. In this paper, we describe the functional requirements in terms of *business goals, actors, use cases, data managed by the system* and *services* (atomic and modular system functions). Non-functional requirements are usually associated with functional requirements (e.g. time-efficiency of a service). If they are not, they should be factorized further (e.g. using MOQARE, see below). The *architectural requirements* describe desired architectural components and architectural constraints. They are often derived by a misuse analysis or by using architectural patterns which translate high-level requirements into more detailed requirements.

In our own work, we use TORE [54] to elicit functional requirements and MOQARE (Misuse Oriented Quality Requirements Engineering) to elicit non-functional requirements. Both also produce architectural requirements. MOQARE is a method for elicitation and documentation of requirements from vaguely defined business and quality goals by a misuse analysis (similar to a risk analysis). This method is described in detail in a technical report [55] and less detailed in a workshop publication [56].

The MOQARE misuse analysis identifies *business goals* of the system and *business damages* threatening them. Potential *misuses* causing these business damages are identified as well as *countermeasures* which detect, prevent or mitigate the misuses. Countermeasures can be all types of requirements. Although these principles stem from the security domain, we could show [55] that they work equally well for all types of non-functional requirements.

To solve requirements conflicts, the requirements must be characterized by several attributes such as their *source, benefit* and *cost, complexity* (*cost*) and *risks.* As not all requirements will be satisfied, we need a *realization attribute* with the possible values "totally/partly/impossible/postponed". They will be explained during the process as they are defined.

Although our process is tailored to the TORE and MOQARE notation of requirements as input, we believe that it can also be adapted to other requirements notations.

4. Types of Requirements Conflicts

The requirements derived by the requirements elicitation are usually conflicting as long as their consistency has not been checked, especially when several stakeholders are involved. Research about requirements conflicts rarely defines exactly what a conflict is, e.g. van Lamsweerde *et al.* [25] observed: "In fact, there is no common agreement on what a conflict between requirements does really mean. The lack of precise definition is a frequent source of confusion." We distinguish three main types of requirements conflicts: requirements inconsistency, requirements contradiction, and feasibility conflict.

Requirements inconsistency: We define requirements to be inconsistent when their conflict can be detected within the requirements space and the solution of this conflict does not signify a decision between solution alternatives. Such inconsistencies might be terminology problems. Several types of requirements inconsistencies are identified by van Lamsweerde *et al.* [25], like inconsistency between different levels of description, if one real-world concept has different names or different structures in the requirements specification, or if one name in the requirements specification designates different real world concepts. Another classification of Grünbacher *et al.* [29] names these types: unclear terms/statement or missing information, incorrect statement, unverifiable statement, ambiguous term. Such inconsistent requirements are specified by different stakeholders [57] or arise due to requirements specification deficiencies. They can be detected by a requirements document review and can be solved within the requirements space, because their solution does not depend on realization considerations. Their detection is supported by grouping the requirements according to the requirements concept(s) they refer to.

Requirements contradiction: Some requirements contradict each other, and the solution of this conflict signifies a decision between solution alternatives. Such a contradiction usually means that two requirements refer to the same requirements concept (e.g. a use case, data group, service, or a concept of the architectural requirements), but demand contradicting values of the same attribute (see also the definition of Egyed and Gruenbacher [47]). Example: "R1: Report X shall show all patient address data" and "R2: Report X shall show only name and postal code" (requirements concept = report X, attribute = report content). Requirements contradictions mostly need to be solved in the solution space, because the decision criteria cost and risk depend on the chosen solution and can only be estimated here. But sometimes, cost and risk are not important. Typically this is the case for standard functionalities like reporting, because they are an essential part of the system and different configurations usually do not differ too much in cost.

Requirements contradictions — like the requirements inconsistencies — can also be detected by a review of the requirements document and by grouping the requirements according to the requirements concept(s) they refer to.

Feasibility conflict: Even requirements which do not conflict in the requirements space, may not be realizable in any of the available architectural solutions at the same time or equally well. We define: "Two or more requirements have a feasibility conflict with each other when they cannot be realized all in the same architectural design." Feasibility conflicts can only be detected when analyzing architectural designs and they demand a decision between alternative solutions. In our case study, such a feasibility conflict occurred among these requirements: "R4: users must be able to edit the items of some of the value lists", "some value lists must not be ordered alphabetically (R5)" and "R3: data which are entered via value lists have to be comparable" although the system offers the value lists in different languages (R7). This conflict was caused by technical constraints of the software chosen.

Not all conflicts mean that the two (or more) conflicting requirements cannot be realized at the same time, i.e. exclude each other completely. Requirements can also conflict partially, if one of them can be satisfied partially. This is possible especially with "imprecise requirements", also called "softgoals" elsewhere, with cross-cutting requirements applying to many concepts, or when the conflict only occurs in special cases. Yen and Tiao [35] define: "A requirement is imprecise if it can be satisfied to a degree." Two imprecise requirements conflict when "an increase in the satisfaction degree of one requirement decreases the satisfaction degree of the other." This applies to contradictions and feasibility conflicts.

5. ICRAD: Steps, Activities and Methods

Requirements negotiation and architectural design mean decision-making, like solving requirements conflicts and choosing that architectural design which satisfies the requirements best. In Sec. 2, we identified several requirements validation, negotiation and architecture design activities, which we now group into seven steps. For each activity, one method was chosen. We set the following criteria for the choice among alternative methods: The input and output must fit the other activities' input and output (i.e. level of detail, notation, etc.). We preferred the most general and simplest method. If needed, it can be replaced by a more specific, more complex method. Our goal was an ensemble of simple, flexible and practicable methods with clearly defined input and output.

Inputs to the ICRAD process are requirements as described in Sec. 3. After several iterations, we get a process output of improved, conflict-free, realizable requirements with a realistic estimation of their feasibility, benefit and cost, plus the architectural design which was the basis for this estimation. This means that the requirements negotiation cannot be finished before the architecture is designed.

Before describing the steps in detail, we give a short overview:

0 — Requirements Specification: We specify the requirements with the methods TORE and MOQARE and get requirements in the form as described in Sec. 3. A first estimation of benefit or identification of mandatory requirements is also necessary to identify the most important (core) requirements later on.

A — Requirements Review and Negotiation in the Requirements Space: Here, dependencies among requirements are detected and documented in the form of requirements bundles (feature bundles, concept bundles). These bundles help to identify requirements inconsistencies and contradictions. Requirements inconsistencies are solved in this step. Some requirements contradictions are solved as well.

B — Identification of Logical Components: The identification of logical components (another bundling) helps to reduce the complexity of the requirements.

C — Design of and Identification of Architectural Alternatives: Here a draft of the high-level architecture is designed. Doing so, open architectural decisions and alternatives are identified. Requirements contradictions also lead to alternatives.

D — Negotiation in the Solution Space: Architectural alternatives are negotiated on the basis of the benefit of the requirements they realize, also considering benefit, cost, complexity and risks of alternatives. Feasibility conflicts are detected here. Finally, the architectural decisions are taken.

E — Review of Design and Identification of New Architectural Alternatives and Open Conflicts: Here, we check whether the architectural decisions taken in step D solved the requirements contradictions and feasibility conflicts, or whether they created new ones. These requirements conflicts then lead to new open architectural decisions. As long as requirements conflicts are open, the process is repeated from step A.

Z — Low-level Design: On the basis of the high-level design, which now realizes conflict-free requirements, the low-level design are designed.

The concepts involved in ICRAD are:

a. requirements
b. core requirements
c. feature bundles
d. requirements concept bundles
e. requirements inconsistencies
f. requirements contradictions
g. logical components
h. complexity of requirements
i. architectural decisions and alternatives
j. architectural bundles
k. feasibility conflicts
l. benefit, cost and risk of requirements
m. high-level design
n. requirements realization + project scope
o. low-level design

Table 1 summarizes the seven steps and which concepts are put in (i) or put out (o) or changed (c) by which step. For instance, the review of the requirements document (A) does not produce the requirements (a), but it will probably change them by solving requirements inconsistencies.

Table 1. Relationships between steps and concepts.

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**
0	o														
A	c	o	o	o	o	o									
B	i	i					o	o							
C	i	i				i	i		o						
D	c	i					i	i	i	o	o	o	o		
E	i	i				o			o		o	i	c	o	
Z													i		o

Now we describe the full process without step 0 and Z.

A — Requirements Review and Negotiation in the Requirements Space

This step includes the review of the requirements, identification of *core requirements*, bundling of requirements according to user view (*feature bundle*) and according to which requirements concept they refer to (*concept bundle*), detection of requirements inconsistencies and contradictions, and finally solution of inconsistencies.

For the **review of the requirements** document, we propose general inspection methods [58] and the criteria defined by the Standard IEEE Std. 830-1998

[22]: correct, unambiguous, complete, consistent, verifiable, ranked for importance and/or stability, modifiable, traceable. For each requirement, the source is required, i.e. the stakeholder who defined it or other sources (like document analysis, misuse analysis).

The requirements are **bundled** according to two criteria:

Feature bundles: From the user point of view, one requirement might make no sense to be realized without another. For instance, the management of a certain type of data makes no sense when they are not reported on. The feature bundle groups together such requirements which from the user point of view only make sense when being realized together.

Requirements concept bundles: Requirements are grouped according to the requirements concept which they refer to, like a use case or a data group.

To illustrate this method, we assume a simple system with three requirements: I, II and III, where I has the highest benefit for the system. Requirements I and II form a feature bundle, and II and III belong to the same requirements concept.

Identification of core requirements: When the system is very complex and is described by many requirements, it makes sense to concentrate on the so-called core requirements during the first iteration(s) of the process. The core requirements are identified by asking the stakeholders, which feature bundles or requirements are mandatory. As feature bundles are requirements which must be implemented together, the core requirements must contain whole feature bundles, even if the stakeholders name single requirements. In our example, the mandatory requirement is I, but as it forms a feature bundle with II, the core requirements are I & II.

We **detect requirements inconsistencies and contradictions** by looking at the requirements concept bundles. We check for the inconsistency types listed in Sec. 4 and whether these requirements demand inconsistent/contradicting values of the same attribute in the same concept. We characterize a conflict by its estimated degree of conflict (total/partially, or in percent).

Solution of requirements inconsistencies and contradictions: Requirements inconsistencies can be solved within the requirements space, as they are caused by a mere problem of wording. Requirements contradictions are also solved in the requirements space, if cost and risk are no decision criteria or can be expected to be approximately equal to the alternative solutions. These conflicts have to be solved by the stakeholders who have originated the inconsistency or contradiction and who are documented as the requirements' sources.

B — Identification of Logical Components

If the system is complex and may not be realizable at the given cost, in the first iteration, only the core requirements are analyzed here and in steps C and D.

This step identifies logical components, which belong to the requirements space. *Logical components* are bundles of dependent data and services (and requirements which refer to them), which serve as a first draft of later architectural components, but are defined independently of any realizable solution. The first step to identify logical components is grouping data and services together to form data groups and service groups. Data groups can be identified on the basis of an entity-relationship diagram of the data managed by the system: Data which belong to the same data group have a 1-1-relationship. Service groups can be defined as a group of services which manipulate or use one data group. Then, starting with the concept bundles referring to data and services, all requirements are attributed to the logical components (data groups or services groups) which they refer to. The **mapping** of requirements to logical components is done in a matrix (for an example see Table 4). Later on, it will help to identify dependencies among requirements. In our example, we assume that three logical components are identified (X, Y, Z): I belongs to X and Y, II to X and Z, and III to Z.

C — Design of and Identification of Architectural Alternatives

This step maps the logical components to architectural components, and identifies architectural decisions and alternatives. Their negotiation follows in step D.

By depicting the dependencies among the logical components in a Design Structure Matrix (DSM) [59], logical components can be grouped together to high-level architectural components. A DSM (Design Structure Matrix) is a quadratic matrix in which all non-zero matrix elements denote a dependency. Different types of dependencies can be supported. The matrix reads (see example in Table 2) like this: "Component Y depends on component X via a dependency of type 1". Dependency types can, for instance, describe whether one component needs data from the other (e.g., dependency of a service group of a data group) or call each other (e.g., dependencies among services).

Table 2. Example DSM for logical components.

	Component X	Component Y	Component Z
Component X	—		
Component Y	1	—	2
Component Z		2	—

The goal is to identify groups of logical components which have high cohesion (i.e., many dependencies) within the group, but low coupling with the other groups (i.e., few dependencies among requirements which belong to different groups). Often, blocks are visible in the matrix or can be created by varying the order of the lines and rows. In Table 2, you can see that Y depends on Z and vice versa. Identifying such blocks of mutually dependent logical components helps to group them into architectural components. Here, Y and Z form architectural component y, and X is mapped to x.

In this step, the **high-level architecture** is (probably only partly) designed top-down. Each time when different choices are possible during the design, this is noted as an architectural decision which is defined by two or more alternatives (i.e. alternative architectural solutions). Here, in step C, they are identified, in step D the alternatives will be compared and the decisions made. Alternatives do not necessarily describe the architectural realization of the whole system, but usually describe alternative choices like "use Oracle" or "use FileMaker". Furthermore, requirements contradictions are considered in this step and can define further architectural decisions.

We use the high-level design concepts according to Bruegge and Dutoit [4]. Their modeling is consistent with the $4 + 1$ view model of software architecture of Kruchten [37]. In the following we set the names according to Bruegge and Dutoit [4] in bold, and add the names according to the $4 + 1$ view in brackets and italic. (Remark: Scenarios and the development view of the $4 + 1$ view are described by the requirements.) The design of the high-level architecture is done top-down in the following order:

The identification of **architectural components** (also called "subsystem decomposition" [4]) is supported by the DSM as described above. Here, architectural styles are chosen such as: layer (e.g. OSI model), repository architectural style, model/view/controller, client/server, peer-to-peer, three-tier, four-tier, pipe and filter (their advantages and disadvantages are discussed by Bruegge and Dutoit [4] on pages 238ff), see also Bengtsson *et al.* [15] and Bass, Clements, Kazman [60]. (*logical view*)

Map architectural components to hardware/software resources and also software to hardware.

Which **persistent data** are **stored** on which component (choose also among flat files, relational database, object-oriented dababase)? (*physical view*)

Design the **global data and control flow** (*process view*). There are three possible control flow mechanisms: procedure-driven control, event-driven control and threads. (Their advantages and disadvantages are discussed by Bruegge and Dutoit [4] on pages 275–277.)

During this top-down design of a system, many decisions have to be taken. High-level decisions define which alternatives one will get on a lower level of design, e.g. the decision for either FileMaker or Oracle has far-reaching consequences. Therefore, the goal of step C is not to identify all decisions at once, but to start with the identification of architectural components, and when all decisions on this level are identified, to proceed to step D and make the decisions. Decisions on lower levels are identified during the following iterations of the process.

In our example, let us assume that when mapping architectural components to hardware, there are the following two alternatives: Alternative 1 supports architectural components x and y with the same hardware, and in alternative 2 one own hardware device is chosen for each of the two architectural components.

D — Negotiation in the Solution Space

After having identified the architectural alternatives in step C, the following activities are performed here: mapping of requirements to architectural components, prioritization of decisions, feasibility check, negotiation between architectural alternatives, requirements update.

Architectural bundles: Requirements depend on each other when they are being realized by the same architectural component. Therefore, in a matrix we map the requirements to those architectural components by which they are being realized, as far as this is known by now. Often it is sufficient and more efficient to do this indirectly by mapping logical to architectural components. These mappings help to identify the requirements which can be realized by an architectural design, but also to show the impact of architectural decisions on architectural components and thereby on logical components and requirements. Dependencies between requirements which are realized by the same architectural component become visible, not forgetting that it supports traceability and change management between requirements and design, during later phases of the system life cycle (not treated here).

In our example, architectural component x influences the logical component X and therefore the requirements I and II. Also, y influences Y and Z, and therefore all three requirements. The decision between alternatives 1 and 2 defined in step C affects both architectural components and therefore all requirements. For larger systems with more requirements, often not all requirements are being affected by a decision.

As some design decisions have a higher impact than others, and as the result of one negotiation will influence the alternatives available for other decisions, it is essential to **prioritize decisions**, and to make the one with the highest impact first. For each decision, the affected architectural components are determined, and these — via the architectural bundling — lead to the affected logical components and requirements. Other decisions affected are those which concern the same architectural component or logical component. The more elements a decision affects, or the more important these are, the higher the priority of the decision. In our example, decisions concerning architectural component y should be made before those concerning x.

To document the prioritization of decisions and the order in which they have been made, the decisions can be sorted in a list in the order of their priority or — if complexity demands it — presented hierarchically in a graph, as Bass *et al.* propose [61]: "At this point we have identified two useful graphs to help to understand why a system is the way it is. The first is a causal graph that shows design as a sequence of decisions, with which we can trace the genealogy of a design decision. The second is a structural graph which presents design as the structure of the software (the result of applying a decision), with which we can trace the genealogy of an architectural element." We suggest to do so for complex systems.

Requirements contradictions have already been identified in step A (requirements review) and in step C architectural decisions have been defined for solving them. As decisions in the solution space usually affect more than one requirement or requirement conflict, we do not solve requirements conflicts in pairs in the solution space, but in part from open architectural decisions and evaluate the alternatives according to all their consequences.

The following feasibility check and negotiation between alternatives will be done for one decision after the other, starting with the most important one. If a decision has significantly changed the architectural design or requirements, then a design review (step E) should be inserted and a new iteration started.

Before describing the feasibility check, we have to make some remarks on the benefit and risk estimation. We estimate the benefit of a requirement on a relative scale of 0 to 3 points. Any other scale is possible, also rating benefit in Function Points or a currency; however, the latter is more difficult. To negotiate architectural alternatives, relative values are sufficient, but they should ideally be comparable to cost in order of magnitude. The benefit measures not only financial value, but also reputation and customer or end-user trust.

Assuming that total benefit, cost and risk of a system depend on all requirements which are being realized by it, the additional benefit, risk and cost in realizing a further requirement depends on the fact of which requirements have already been realized before. In a recent work [62], we discuss that benefit, cost and risk of a given requirement are no fixed requirement attributes, which can be estimated once for all times, but must be estimated relatively to a reference system design, which is clearly defined by the requirements which are being realized by it. A requirement's benefit relative to a reference system is equal to the gain in system benefit, when this requirement is being realized additionally, and its cost the corresponding additional cost. Benefit, cost and risk are not summable, i.e. the total benefit of a system is not equal to the sum of the individual requirements' benefits.

The benefit is estimated top-down, starting with the business goals. We distinguish three cases to estimate the benefit of single requirements:

- Some requirements were requested explicitly by the stakeholders, because they directly support the business goals. Their benefit is estimated by asking which benefit — relative to these business goals — would be gained, if this requirement was being realized. This benefit is reduced by risks of misuses which threaten the benefit of this requirement (risk estimation is described below).
- Some requirements support other requirements (e.g., non-functional requirement on use case or service or architectural constraints). Their benefit is estimated by asking: How much benefit does this non-functional requirement add to the use case or service or architecture, if being realized? Its benefit is also reduced by risks. If a use case has benefit 2.0 with the performance demanded, but only 1.0 without, then the benefit of its performance requirement is 1.0.
- Some requirements are countermeasures, i.e. they detect, prevent or mitigate a

misuse. We use risk estimations to estimate a countermeasure's benefit. A misuse has causes and consequences. Using the MOQARE concepts, we say that the causes of a misuse are a misuser A and maybe a vulnerability B. These might lead to the threat C (an action) to take place, and this eventually leads to damage D, which means that the benefits of requirements or satisfaction of business goals are threatened. This loss of benefit is called $l(D)$. The risk of a misuse is measured by *probability times damage* (e.g., see ISO standard [63]). According to the rules of probability calculation, this makes:

$$risk = p(A \cap B) \cdot p_{A \cap B}(C) \cdot p_{A \cap B \cap C}(D) \cdot l(D) \quad (1)$$

The conditional probability $p_{A \cap B}(C)$ denotes the probability that threat C happens if both A and B are given. The estimations of the probabilities and the expected damage depend on whether countermeasures are supposed to be realized in the reference system.

The benefit of a countermeasure is equal to the misuse's risk if the countermeasure prevents the misuse totally. Otherwise it is proportional to its effectiveness. If the countermeasure reduces the probability of the misuse by 30%, then the benefit of the countermeasure is 30% of the misuse's risk.

The **feasibility check** evaluates which requirements can be realized by which alternative, at which cost, complexity and risk. Here, only those requirements which are affected by this decision are analyzed, and for complex systems only the affected core requirements. The reference system can be different for each decision, as it is modified by the decisions made before. For each architectural alternative, the following values are estimated by an architecture specialist:

- *Realization* of the requirements concerned by the alternative; values of this attribute are: "totally/partly/impossible/postponed".
- *Benefit* of the alternative, which is not equal to the sum of the benefits of the realized requirements, but depends on which requirements or feature bundles are being realized totally or partly or not at all. The benefit of the alternative is estimated as the benefit added to the reference system by choosing this alternative, estimated on a scale of 0 to 3 points.
- The *risk* of an architectural alternative summarizes different types of misuses:
  - — Misuses, provoked by risky requirements which are being realized by this alternative or misuse, provoked by the architectural solution described by the alternative.
  - — Misuses, provoked by not realizing some requirements.

  Risks are calculated from probability and damage estimates as defined in Eq. (1). The risk of an alternative is only measured by the sum of the risks of all these misuses, if they are independent. Dependent risks must be treated as one.
- *Cost* of implementation. It must be estimated in the same unit as the benefit. The alternative's cost is the additional cost caused by implementing it additionally

to the reference system.

- *Complexity* includes architectural and organizational complexity and will lead to maintenance and other costs which add to the implementation cost. For being comparable to the other criteria, complexity must be transformed into complexity cost. Complexity metrics usually predict maintenance cost and therefore a defined period of time is needed for this transformation, e.g. the expected lifetime of the system or the planned time until breakeven. Usually, architectural analysis methods measure the complexity of an architectural design by the strength of the coupling of its components, i.e. by stating how many requirements are supported or influenced by one component and how many components are affected by one requirement (see SAAM [45, 46] and SAAMCS [63]). The complexity of the integration of an IT system into its environment can also contribute to the maintenance cost.
- **Feasibility conflicts** of requirements (definition see Sec. 4). Conflicts are characterized by their degree of conflict. Conflicting requirements can either be mutually exclusive (conflict degree 100%) or partially conflicting.

It is not easy to estimate these values on the basis of an architectural draft. Bosch and Molin [5] name four approaches to assess how well requirements are being realized by an architecture: scenarios, simulation, mathematical modeling and experience-based reasoning. We leave it to the architecture specialist to choose the right method for the feasibility check.

The feasibility check is done for both (all) alternatives of the same decision. In our example, alternative 1 has benefit B1, cost C1, complexity cost CC1 and total risk R1, Alternative 2 is described by benefit B2 (here higher than B1 assuming requirement II is realized better), but cost C2 and complexity cost CC2 are higher because more hardware is needed and there is an additional interface between the two hardware devices. This interface can also provoke additional risk, so finally risk R2 > R1. Now, how to decide? If the more expensive solution has a lower benefit, then it is logical to choose the cheaper and better solution. However, very often, the alternative with the higher benefit is the more expensive one, as is the case in this example.

To **negotiate architectural alternatives**, means to evaluate the alternatives with one or more decision criteria and then to choose the most favorable alternative. Yen and Tiao [35] describe a negotiation like this: "...we should explore a feasible requirement [here: architectural design] that maximizes the overall degree of satisfaction." Possible factors of such a satisfaction value are benefit, cost, complexity and risks, which we determined by the feasibility check.

To combine these factors, several derived values are calculated: the total benefit and total cost of an alternative, its net value and benefit-cost-ratio. We also calculate these values for the difference between two alternatives, plus the ratio of the benefit and cost differences We refer to formulae from the SQUARE project [53] and CBAM [43, 44].

Usually, one defines the **total benefit** of a system to be the sum of all benefits of all realized requirements minus the risks (see Eq. (2)). The **total cost** must include the complexity cost (see Eq. (3)).

$$\text{Total benefit} = \Sigma\ (\text{benefit of realized requirements — risks}) \tag{2}$$

$$\text{Total cost} = \Sigma\ (\text{cost of realized requirements} + \text{complexity cost}) \tag{3}$$

Here, we must remind ourselves that strictly speaking benefits, risks and cost must not be summed to calculate total benefit and total cost. However, it is an approximation frequently used, and as we have no equally simple formula to offer, we use it, keeping in mind that, when necessary, dependencies among requirements and risks must be taken into account, e.g. by estimating the benefit of a whole feature bundle or treating dependent risks as one.

From total benefit and total cost we derive two further satisfaction criteria:

- the **net value** of an alternative = *total benefit* minus *total cost*
- **Benefit-cost-ratio** = *total benefit/total cost*

Both criteria have their advantages and disadvantages, and therefore we use them both. Advantage of the net value: Additional risks reduce the benefit of an alternative [43, 44, 53], but one also could say that they increase the expected cost. The net value does not depend on whether risks are counted on one or the other side. In our example, alternative 2 represents requirement II better than alternative 1, but also has additional risk $\Delta R = R2 - R1$ and additional cost $\Delta C = C2 - C1$. The net value of alternative 2 relative to alternative 1 is $(\Delta B - \Delta R) - \Delta C$, assuming that risk reduces the total benefit. If we say that risk adds to the total cost, we get $\Delta B - (\Delta C + \Delta R)$, i.e. the same value. The benefit-cost-ratio, though, leads to different values: $(\Delta B - \Delta R)/\Delta C$ or $\Delta B/(\Delta C + \Delta R)$.

Advantage of the benefit-cost-ratio: We work with estimated values with an arbitrary unit of measure. The orders of magnitude of cost and benefit can be different in scale, if 3 cost points do not correspond to 3 benefit points. This can be neglected when comparing benefit-cost-ratios, as the relative error (expressed by a multiplication factor) is the same for the ratios of all alternatives.

To document the **negotiation of two or more alternatives**, we introduce the template which is shown in Table 3. The value $[(B2-R2)-(B1-R1)]/[(CC2-CC1)+(C2-C1)]$ in the table field on the lower right (= ratio of the total benefit and total cost differences) is independent of both of the effects discussed above. To interpret this value, different cases have to be distinguished (we write ΔTB for the difference in total benefit, i.e. total benefit of alternative 2 minus total benefit of alternative 1, and ΔTC for the difference between the total costs of the alternatives, so this value is written as $\Delta TB/\Delta TC$):

- If $\Delta TB/\Delta TC < 0$, a clear decision can be taken:
  - If $\Delta TB < 0$ and $\Delta TC > 0$, then the total benefit of alternative 1 is higher and total cost below that of alternative 2, and alternative 1 is chosen.

  - If $\Delta TB > 0$ and $\Delta TC < 0$, then the opposite is true and alternative 2 is chosen.

- If $\Delta TB/\Delta TC > 0$ because both ΔTB and ΔTC are positive
  - and the absolute value of $\Delta TB/\Delta TC > 1$, then alternative 2 is chosen;
  - and the absolute value of $0 < \Delta TB/\Delta TC < 1$, then the alternative with the higher benefit-cost-ratio is chosen.
- If $\Delta TB/\Delta TC > 0$ because both ΔTB and ΔTC are negative
  - and the absolute value of $\Delta TB/\Delta TC > 1$, then alternative 1 is chosen;
  - and the absolute value of $0 < \Delta TB/\Delta TC < 1$, then the alternative with the higher benefit-cost-ratio is chosen.

These three criteria (net value, benefit-cost-ratio and ratio of the benefit and cost differences) do not always lead to the same decision. If they are in favor of different decisions, it has to be decided which satisfaction criterion should be maximized and also it has to be checked which of the effects discussed above might have the stronger falsifying influence on the result here. If for instance the order of magnitude of cost and benefit are very different and the net value of an alternative can be wrongly negative, then the benefit-cost-ratio or $\Delta TB/\Delta TC$ should be the decision criterion.

Another question which occurred with the use of this template is: If a requirement is not satisfied by alternative 1, does it then reduce the benefit directly or add to the risk? As we work with the total benefit, which is the difference between benefit and risk, this decision makes no big difference, but it has to be made because otherwise one might count the same risk twice. Therefore we decide: When the non-realization of a requirement means a direct and inevitable loss, it is subtracted from the benefit, but when this non-realization produces a risk (i.e. something that might happen with a certain probability), then it counts on the risk side.

Table 3. Template table used to compare alternatives.

	Alternative 1	Alternative 2	Difference
Cost	C1	C2	C2 − C1
Complexity cost	CC1	CC2	CC2 − CC1
Risk	R1	R2	R2 − R1
Benefit	B1	B2	B2 − B1
Total benefit	B1 − R1	B2 − R2	(B2 − R2) − (B1 − R1)
Total cost	C1 + CC1	C2 + CC2	(CC2 − CC1) + (C2 − C1)
Net value	(B1 − R1) −(C1 + CC1)	(B2 − R2) −(C2 + CC2)	(B2 − R2) − (C2 + CC2) −(B1 − R1) + (C1 + CC1)
Total benefit/ total cost	(B1 − R1)/ (C1 + CC1)	(B2 − R2)/ (C2 + CC2)	[(B2 − R2) − (B1 − R1)]/ [(CC2 − CC1) + (C2 − C1)]

In addition to this table, the alternatives and details of their cost and risk, benefit and complexity estimations must be documented in detail elsewhere, also the architectural components and requirements affected (as has been done in the case study in Sec. 6). So later on, when the conditions or information have changed, the rationale of the decision can be reproduced or questioned.

Requirements update: The architectural decisions can lead to new requirements and risks, which now have to be included in the requirements documentation. If they only apply to one specific architectural alternative among several, they must be marked accordingly. (We call such requirements, which only belong to one architectural alternative, *induced requirements*.) Those requirements, which refer to an alternative which has been rejected, can now be discarded.

E — Review of Design and Identification of New Architectural Alternatives and Open Conflicts

A review of the design is performed by developers or designers who were not involved in the design process. They check the following criteria on the design document: Is it correct, complete, consistent, realistic, readable? (More detailed questions can be found on page 281f of Bruegge and Dutoit [4]). These criteria must also be fulfilled for the relationship between design and requirements: It must be possible to map the design to the requirements. This means that for each architectural component, there is at least one requirement, and each requirement is addressed. This includes the architectural requirements.

Not only is the design document reviewed, but also the design itself is reevaluated: Have all four architecture levels, as defined in step C, been designed? Does it meet the business goals and core requirements? The requirements realization attribute is reviewed which indicates whether a requirement will be satisfied "totally/partly/impossible/postponed", and also the project scope is identified. Which is the total cost of the designed architecture? Which requirements conflicts could be solved and which are still open?

During this review, new requirement conflicts and architectural alternatives can be detected, which lead to new architectural decisions, which might lead to an improvement of the architectural design. Based on this review, it has to be decided whether a new iteration is necessary and the process then starts again with step A.

In our example, we might have decided in favor of alternative 2, which has higher risks. We can now define countermeasures against these risks, which are new requirements. If these countermeasures are in conflict with another requirement, a new negotiation and decision among the system as designed so far or an alternative 3 (realization of the countermeasure and renouncement of the requirement conflicting with the countermeasure) will be necessary.

6. Case Study

A case study was performed to illustrate the requirements validation, negotiation and architectural design process by a realistic example. The "Uveitis Database" is used at the Interdisciplinary Uveitis Center Heidelberg. Ophthalmologists and internists work together to diagnose and treat the non-trivial causes of Uveitis, an inflammatory eye disease. The Uveitis Database manages patient admission data (name, address, date of birth, insurance and insurance number), as well as their examination results, diagnosis, medication and surgery data at different points of time, for the analysis of the therapy course and as a database for scientific studies. The system uses the software FileMaker. It is in operative use and available in several languages.

This case study was performed by a specialist for the suggested negotiation process and the software engineer managing the Uveitis Database.

During this case study, we discussed three new requirements which are relevant for further enhancement of the system and which are suitable to demonstrate our process and methods. The new requirements are:

- R1: Use the web client of FileMaker instead of or additionally to the client software.
- R2: A card reader is to be used for automated input of admission data.
- R3: The data which are entered via value lists in different languages are to be comparable.

To provide a basis for this case study, first the existing system was analyzed, identifying requirements bundles, logical and architectural components.

A — Requirements Review and Negotiation in Requirements Space

As input for the requirements review we used the requirements resulting from an earlier case study on the Uveitis Database. A review of the requirements had already been carried out to check their consistency.

Bundling according to requirements concept: Use cases have been bundled according to their actor and according to the data which are entered, managed or reported in each use case. Each use case could be assigned to exactly one of the six actors, so this bundling leads to six disjoint bundles. Several use cases, though, manipulated more than one of the five identified data groups, so there are five overlapping bundles. R2 is attributed to actor "reception" and to data group "admission data". R1 and R3 refer to all actors, all data and all use cases.

Feature bundles, from the users' point of view: Before the Uveitis Database was implemented, the whole process of patient examination at the Uveitis Center was supported by paper templates or other systems (e.g. calendar, SAP ISH MED). As the paper documentation is still used in parallel to the database, the use cases can be treated independently. If a use case is not implemented, it can still be supported by the actual system. There is only one exception: the reports. If certain data are managed in the system, then a corresponding report also has to be provided. The feature bundles therefore consist of the use cases referring to each type of data (patient admission, examination, and surgery) plus the corresponding report functionality plus all related requirements, e.g. non-functional. The card reader requirement R2 belongs to the feature bundle referring to the management of patient admission data. R1 and R3 are cross-cutting requirements which concern all features but do not need to be attributed to any feature bundle as they make sense independently of the others.

As the requirements of the existing system have been reviewed in an earlier case study, new **inconsistencies** and **contradictions** can only appear between the former requirements and the new ones or among the new requirements. An inconsistency might arise when the same terminology is not used for the new requirements as for the existing requirements, e.g. when the specification says that the card reader is used for automated input of "admission information" instead of "admission data". Such an inconsistency would be identified by the above bundlings. Either when attributing R2 to the corresponding feature bundle or requirements concept bundles or at the latest when looking at the bundles during review, the inconsistency would have been detected.

There is one requirements contradiction among R3 and an older requirement which is "R4: Users must be able to edit the items of some of the value lists". If the users edit their own value lists, then the comparability of data entered in different languages can probably no longer be guaranteed. Both requirements refer to the data group "value lists" and therefore belong to the same requirements concept bundle. The conflict is partial. Its degree cannot be estimated in numbers here, but needs more architectural knowledge which is not available in the requirements space.

B — Identification of Logical Components

For the Uveitis Database, eleven **logical components** — comprising five data groups and six service groups — were identified from the requirements. Table 4 represents the mapping of some of the requirements (left column) to the logical components by which they are realized or which they affect. To illustrate this, we include R4 and some more of the old requirements (on different level of granularity), which have also been relevant in some of the negotiations in step D.

As one can easily see from the table, the complexity of some requirements — measured by the number of logical components concerned is very high (like "R9: authorization concept" or "R12: data integrity"), while others only apply to one or few logical components (R2: card reader). As the list does not contain all requirements, the complexity of the logical components (i.e. the number of requirements they realize) cannot be determined based on Table 4.

C — Design and Identification of Architectural Alternatives

The identification of architectural components starts with grouping the logical components which were identified in step B. This had already been done during the

Table 4. Case study: mapping of requirements to logical components.

	d. gr. admission	data group exam	data g. surgery	d.gr. value lists	d.gr. user admin	s.gr. admission	service gr. exam	s.gr. surgery	s.gr. internist	s.gr. reports	s.gr. user admin
R1: Use the web client of FileMaker instead of or additional to the client software.				x		x	x	x	x	x	x
R2: A card reader has to be used for automated input of admission data.	x					x					
R3: Data which are entered via value lists have to be comparable.				x							
R4: Users must be able to edit the items of some of the value lists				x		x	x			x	
R5: Some value lists have to be ordered alphabetically, some must not.				x							
R6: use case "manage admission data"	x					x					
R7: the user interfaces have to be available in German as as well in English and in further languages	x	x		x		x	x			x	
R8: user administration					x						x
R9: authorization concept	x	x	x		x	x	x	x		x	x
R10: usability						x	x	x	x	x	x
R11: availability of user interface						x	x	x	x	x	x
R12: data integrity	x	x	x		x	x	x	x	x	x	x
R13: logging of data changes	x	x	x		x						

original design of the system, but we sketch it here to illustrate the principle. Table 5 shows the DSM for the logical components, i.e. their dependencies among each other. A non-zero matrix element ij means that the logical component of row i depends on the logical component of column j. Dependency type 1 means that a data group is relational dependent on another, type 2 means that a service group needs data from a data group, type 3 means that a service is called by another.

The DSM shows that there is a block of filled matrix elements which indicates that it makes sense to group the data (groups) into one architectural component (here: database). It is clear from the DSM that the service groups depend on the data groups and not vice versa. In this case study, the two top-level architectural components are a server and a client.

Table 5. Case study: DSM for logical components.

	data gr. admission	data group exam	data gr. surgery	d.gr. value lists	d.gr. user admin	s.gr. admission	Service gr. exam	s.gr. surgery	s.gr. internist	s.gr. reports	s.gr. user admin
d. gr. admission	----			1	1						
data gr. exam	1	----		1	1						
d.gr. surgery	1	1	----	1	1						
d.gr. value lists				----							
d.gr. user admin					----						
s.gr. admission	2			2	2	----					
s.gr. exam		2		2	2	3	----				
s.gr. surgery			2	2	2	3		----			
s.gr. internist		2		2	2	3			----		
s.gr. reports	2	2	2	2	2					----	
s.gr. user admin					2						----

Design of High-Level Architecture and Identification of Architectural Alternatives: The realization of the three requirements R1 to R3 can lead to changes in the architecture. The web client (R1) means to add further software, the card reader (R2) to add a further hardware device plus the corresponding software. R3 and the requirements contradiction detected in step A question the architecture as it has been so far, especially the realization of the logical component "data group value lists".

D — Negotiation in the Solution Space

Figure 1 shows a decision tree of the design decisions previously made for the Uveitis Database. The decisions were made top-down, and we present the final decisions on the very left branch. Left of the tree we name the corresponding step in the architectural design. The choice of FileMaker also meant that a relational database with procedure-driven control has been chosen, so there was no architectural decision to be made on this level. On a lower level, we can say that the logical components describe architectural components, i.e. the data groups describe database tables and the service groups parts of the application.

The new architectural decisions to be made with reference to the requirements R1–R3 (see last paragraph of step C) are now prioritized and ordered according to their impact on architectural and logical components respectively. For each decision, we name the alternatives and — in brackets — the architectural respectively logical components which are affected:

(1) Realization of value lists: either as lists which are editable by normal users (which lead to non-comparable entries) or defined fixed value lists including translations in separate tables (which are comparable but editable only by

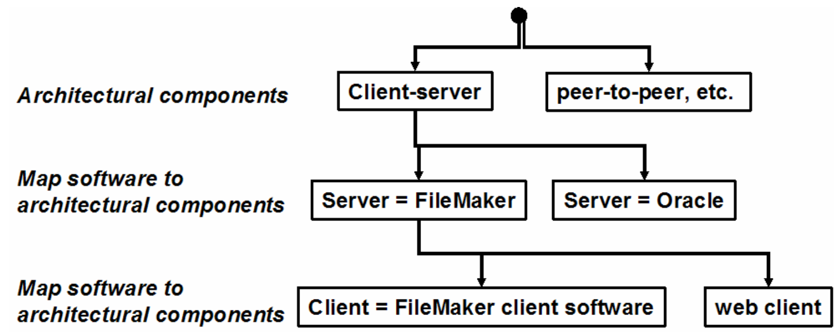

Fig. 1. Case study: Decision tree.

administrators) (see Table 4: affects data group "value lists", three service groups and requirements R3, R4 and R5, the other data groups only indirectly (see Table 5)).

(2) Introduction of card reader or not (see Table 4: affects requirement R2 and consequently data group "admission" and service group "admission" and indirectly (see Table 5) two more data groups and four service groups).

(3) Access via web interface or via client software or both (affects all service groups, but only data group "value lists").

These decisions are independent of each other, i.e. the choice of one or the other alternative does not affect the cost, benefit or risks of the other decisions. Therefore, a similar reference system can be used for all three of them. This reference system is the existing system, using no card reader, giving access by client software exclusively and realizing value lists as lists which are editable by normal users.

One feasibility conflict which we found was the conflict between requirement R3 and R5. When in decision (1) we choose the second alternative, all value lists will be ordered alphabetically automatically. As R5 states that some (actually 7%) of the value lists must not be ordered alphabetically, the conflict degree is 7%.

As an example, we would like to present the details of the negotiation of the decision (3): FileMaker allows an access via web interface or via client software or both in parallel, but the alternative presently used is the client software. Cost, risks, complexity cost and benefits are estimated in an arbitrary unit within the range of 0 to 3 points. The period of time chosen to estimate time-dependent values like maintenance cost is one year. Decision (3) affects requirement R1 and all requirements which depend on any service (see Table 4), but it makes a difference in terms of realization, cost, risk and benefit for only some requirements. When estimating these values for the alternatives, they are not evaluated for the whole system, but only those requirements are considered, where the decision makes a difference. It is assumed that the client software is already in use.

Feasibility check: Using the client software for accessing the database is more

Table 6. Negotiation between the access via client software, web interface and the use of both in parallel.

	Alternative 1: Client software	Alternative 2: Web interface	Alternative 3: Both	Difference between alternatives 2–1
Cost	1.0	0.5	1.7	−0.5
Complexity cost	0	0	0.5	0
Risk	0.15	0.6635	0.8418	0.5135
Benefit	3.0	3.0	3.0	0
Total benefit	2.85	2.3365	2.1582	−0.5135
Total cost	1.0	0.5	2.2	−0.5
Net value	1.85	1.8365	−0.0418	−0.0135
Total benefit/ total cost	2.85	4.6750	0.9810	1.027

secure and user-friendly and is the alternative realized so far, but the requirement "R11: availability of user interface" is better realized when we use the web interface access. Table 6 summarizes the benefits, costs, and risks, as well as the net value and benefit-cost-ratio of the alternatives of

(1) access via client software,
(2) via web interface (via intranet; via internet also would have been possible, but will not be discussed here), that
(3) both are offered in parallel, and the difference between the first two.

The costs of the alternatives 1 and 2 consist of the following factors: The installation costs are assumed to be 1 for the client software (e.g. licence costs), but only 0.1 for the web interface, as a web browser is supposed to be installed on almost all personal computers. Here, only patches are necessary. The use of the web interface would demand an adaptation of the user interfaces, which so far are optimized for the client software access, at the cost of 0.4. For instance, the web interface (in FileMaker Pro version 7.0) supports no pop-up windows. We do not consider the costs for the maintenance of the user interfaces here (we count them on another budget), except for alternative 3 where both access alternatives are supported in parallel, because here the maintenance effort it raised, let us say by the cost of 0.3. Alternative 3 also demands the adaptation of the user interfaces (cost = 0.4) plus an installation cost of 1.0.

The following risks were originally identified and estimated by MOQARE in the earlier case study, but updated now, based on the current architectural knowledge:

- Alternative 1 (client software) enhances the risk that data are not being entered due to lack of availability of the system at a work place. We rate the severity of this loss of data at 3.0 and estimate the probability that this happens with 5%, i.e. this risk is rated at 0.15.

- Alternative 2 (web access) introduces security risks and security measures. As the access is planned via intranet (which is supposed to be secure), and not via internet, the additional security risk of unauthorized use due to the introduction of the web access is estimated to be 0.01 due to the low probability of security incidents.
- Even after the adaptation of the interfaces to the browser, some functions cannot be offered via web interface. This is the choice of multiple values within a value list. The severity of the benefit loss by missing data is rated at 1.5. This risk affects one-fifth of all data fields at most. If we estimate that half of these data will be entered at another work place later, this risk is assessed by the risk $1.5 \times 0.2 \times 0.5 = 0.15$.
- The web interface is visually less attractive and slower, but we consider this to be only a visual problem which has a risk of 0.0035.
- On the web interface, after the input of data, it is necessary to press an additional "enter" button to save, what the users need not do when using the client software. This means that data can be lost, especially in the first phase after the change. We rate this risk at 0.5.
- When using both systems (alternative 3), the difference in the usage of both interfaces can lead to an additional usability problem which we rate to be 0.5. As for the other risks observed for the web interface, we add the security risk of 0.01, but consider the usability and other risks only to be half their values, because the users can choose among the two interfaces (=0.3318), what leads to a risk sum of 0.8418.

In terms of maintainability, alternatives 1 and 2 are approximately equal but for alternative 3 of both clients we estimate additional complexity costs of 0.5. We estimate the benefit of alternatives 1 and 3 to be 3.0, but also for the web interface alone (alternative 2), although multiple choice is not fully possible for one-fifth of the data fields. However, this was already included in the risk estimation.

The decision here is not evident, as the client software (1) has the higher net value, but the lower benefit-cost-ratio than the web interface alternative; (2) The ratio of the benefit and cost differences is 1.027. It is positive because both differences are negative, and its value is >1. This means a decision for alternative 1.

Using both in parallel combines maximum cost with high risk and therefore has the lowest net value as well as the lowest benefit-cost-ratio. Alternative 3 is not favorable.

A compromise could be to offer the web interface only to a few very experienced users who work on changing work places (alternative 2a, not presented in Table 6). This makes sense because usability problems contribute a major part to the risks, and they will be less with experienced users. On the other hand, users with changing work places will introduce costs for installing the client on each of these computers. A re-evaluation of the web interface for such users leads to a risk of 0.1635 (the "enter button risk" was considered to be zero). The result is more favorable for alternative 2a than for 2: The net value of 2.3365 lies above that of the client software, and the benefit-cost-ratio is as high as 5.6730. The ratio between differences in benefit and differences in cost (field on the lower right of the table) is now 0.027 instead of 1.027 before, i.e. a decision for alternative 2a.

The final choice was alternative 2a: to use the client software per default and to introduce the web interface for some experienced users.

Detailed negotiations concerning decisions 1 and 2 were performed but are not described here in detail, only summarized.

Decision 1: Two alternatives concerning the realization of the multi-lingual value lists were compared. The new proposal leads to high benefits and risk reductions which were even strong enough to justify the high costs of realizing all value lists anew.

Decision 2: Use a card reader for automated input of admission data from the insurance card or not? The card reader itself did not add much value to the system, but when it was combined with an automated search of whether this patient is already contained in the system (alternative 3), this solution showed a high value, as the creation of doublets had shown to be an important risk which causes significant loss of data integrity and costs for data cleansing.

Requirements update: As it was decided to allow an additional access via web interface and intranet for some users, this changes the requirements on the user interface as well as the requirements on user training. As these new requirements only apply if a web interface is offered, they must be linked to the alternative accordingly. This allows the deletion of these additional requirements if later on new decisions lead to giving up the web interface.

E — Review of Design and Identification of New Architectural Alternatives and Open Conflicts

There have been several changes to the system architecture (new card reader, allow web interface for some users, different realization of the multi-lingual value lists), and therefore the consistency of the requirements must be checked anew. This demands a further iteration.

7. Conclusion

The goal of this work was to integrate the activities of the solution of requirements conflicts and of architectural design. A clear distinction was made between the requirements space and the solution space, and it was important to consider dependencies among requirements, especially the dependency between architectural design and requirements negotiation: Most requirements conflicts can only be solved in the solution space. Therefore, the process of requirements negotiation must also consider architectural design aspects. Based on this idea, ICRAD (Integrated Conflict Resolution and Architectural Design) was developed. The ICRAD process can be used for the development of a new system from scratch as well as for the enhancement of an existing system (like in our case study).

In this work, three main types of requirements conflicts are considered, as well as nine types of dependencies among requirements and between the requirements space and the solution space. The conflict types are requirements inconsistencies, requirements contradictions and feasibility conflicts. We considered the following dependencies:

- Among requirements when one requirement refines or realizes another. (These are detected during the requirements elicitation in TORE [54] and MOQARE [56].)
- Among requirements which refer to the same requirements concept (requirements concept bundles), as they are potentially inconsistent or contradicting.
- Among requirements which from the users' point of view depend on each other, i.e. only make sense when being implemented together (feature bundles). They must be considered when solving requirements conflicts.
- Among requirements which refer to the same data groups or service groups (logical components) in the requirements space.
- Among requirements which are realized by the same architectural component (architectural bundles). Then we know which requirements will be affected by a decision.
- Between architectural decisions and requirements: The restrictions of architectural alternatives/designs lead to restrictions in requirements and to requirements changes, e.g. when countermeasures against a misuse are not practicable, alternative countermeasures for the same misuse can be considered.
- Between requirements and chosen architectural alternatives (induced requirements), i.e. the bundling of some requirements which are valid for one architectural alternative only. For instance, some architectural designs introduce risks and therefore should only be used under constraints. This leads to new, more detailed requirements which only apply to a specific architectural alternative, e.g. the configuration of an architectural component or interface.
- Between chosen architectural design and negotiation of requirements conflicts: Decision criteria like costs and complexity of a requirement depend on its realization. Therefore, most conflicts cannot be solved in the requirements space, but only on the basis of at least an architectural draft.
- Among architectural decisions: Decisions are prioritized according to their impact (e.g. on other decisions), and the most important is treated first.

The process of requirements negotiation and architectural design was split into activities according to the methods chosen. The bundling and grouping of requirements reduces complexity and leads to an easier handling of the requirements during the process.

A negotiation template was developed which supports and documents the negotiation among architectural alternatives in the solution space. This template well allows the comparison of different alternatives in terms of cost, complexity, benefit and risk and gives an overview of the reasons for the decisions. The method helps to compare different decision criteria, e.g. when high benefit of an alternative is combined with high complexity, high cost, but lower risk than another alternative and therefore the decision would not be evident. It was important to document not only the total benefits and costs of an alternative but also their factors, so when discussing the result, the rationale of the negotiation is still visible, can be questioned and corrected if necessary. Furthermore, it can help to improve solutions.

In our case study, three negotiations were performed. Many parameters played an important role for the estimations, like the number and type of users, the time which is assumed to be the break-even time of the system (for comparing one-time development cost to constant maintenance cost), the stability of the requirements. Where such parameters were not known, the negotiation was performed twice, testing different values. Sometimes, the two alternatives had a significant influence on the feasibility of other requirements which referred to the same logical system component. Such dependencies had been identified and documented in a new negotiation. The high number of assumptions influencing such an estimation supports our assumption that cost and benefit are not fixed attributes of requirements but must be estimated based on a reference system. Furthermore, clear definitions are necessary.

Our idea of solving requirements contradictions and feasibility conflicts by making architectural decisions was successful in our case study.

8. Future Work

More sophisticated case studies will have to be performed, especially those starting from scratch, so we can show whether the early phases of a real project can be supported well by the ICRAD process.

There still remains a risk when comparing benefit to cost. They may systematically be of a different scale. We use both the net value and the benefit-cost-ratio for the negotiation of alternatives. The ratio of the benefit and cost differences between these alternatives served as a good criterion to test which alternative is more favorable. The latter two criteria partly compensate for the scale effect. In our future work, we want have a closer look on how to scale the cost and benefit (order of magnitude).

Further steps can be included into ICRAD like the trade-off of project scope with budget constraints, the estimation of the total project cost, the price negotiation and the decision whether the project is being realized. We are currently investigating these process extensions.

Acknowledgments

The authors want to thank Prof. M. Becker of the Interdisciplinary Uveitis Center Heidelberg for his friendly support.

References

1. E. R. Poort and P. H. N. de With, Resolving requirement conflicts through non-functional decomposition, in *Proc. 4th Workshop Conf. on Software Architecture* (*WICSA*) (2004), pp. 145–154.
2. I. Sommerville, *Software Engineering*, 6th ed. (Pearson Education, 2001).
3. W. N. Robinson, S. D. Pawlowski, and V. Volkov, Requirements interaction management, *ACM Computing Surveys* **35**(2) (2003) 132–190.
4. B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering — Using UML, Patterns, and Java* (Prentice Hall, NJ, 2004).
5. J. Bosch and P. Molin, Software architecture design: Evaluation and transformation, in *Proc. Conf. and Workshop on Engineering of Computer-Based Systems ECBS* (1999), pp. 4–10.
6. A. van Lamsweerde, From system goals to software architecture, in *Formal Methods for Software Architectures*, eds. M. Bernardo and P. Inverardi (Springer-Verlag, 2003), pp. 25–43.
7. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and S. J. Carriere, *The Architecture Tradeoff Analysis Method*, Technical Report CMU/SEI-98-TR-008 (Software Engineering Institute, Carnegie Mellon University, 1998).
8. N. S. Rosa, G. R. R. Justo, and P. R. F. Cunha, A framework for building non-functional software architecture, in *Proc. ACM Symposium on Applied Computing* (2001), pp. 141–147.
9. M. Brandozzi and D. E. Perry, From goal-oriented requirements to architectural prescriptions: The preskriptor process, in *Proc. From Software Requirements to Architectures Workshop STRAW* (2003), pp. 107–113.
10. B. A. Nuseibeh, Weaving together requirements and architectures, *IEEE Computer* **34**(3) (2001) 115–117.
11. A. Egyed, P. Grünbacher, and N. Medvidovic, Refinement and evolution issues in bridging requirements and architecture — The CBSP approach, in *Proc. From Software Requirements to Architectures Workshop STRAW* (2001).
12. L. Xu, H. Ziv, D. Richardson, and T. A. Alspaugh, An architectural pattern for non-functional dependability requirements, in *Proc. 4th Workshop on Architecting Dependable Systems* (*WADS*) (2005), pp. 1–6.
13. F. Losavio, Quality models to design software architecture, *J. Object Technology* **1**(4) (2002) 165–178.
14. J. J. Pauli and D. Xu, Threat-driven architectural design of secure information systems, in *Proc. 7th Int. Conf. on Enterprise Information Systems* (2005), pp. 136–143.
15. P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, *Analyzing Software Architectures for Modifiability*, Working paper, submitted for publication 2000, http://citeseer.ist.psu.edu/bengtsson00analyzing.html.
16. L. Dobrica and E. Niemela, A survey on software architecture analysis methods, *Trans. Software Eng.* **28**(7) (2002) 638–653.
17. R. Kazman, L. Bass, G. Abowd, and M. Webb, SAAM: A method for analyzing the properties of software architectures, in *Proc. 16th Int. Conf. Software Engineering* (1994), pp. 81–90.
18. B. Tekinerdogan, ASAAM: Aspectual software architecture analysis method, in *Proc. 4th Working Conf. on Software Architecture* (*WICSA*) (2004), pp. 5–13.
19. R. Kazman, M. Klein, and P. Clements, *ATAM: Method for Architecture Evaluation*, Technical Report CMU/SEI-2000-TR-004 (Software Engineering Institute, Carnegie

Mellon University, 2000).

20. M. Barbacci, R. Ellison, A. Lattanze, J. Stafford, C. Weinstock, and W. Wood, Quality Attribute Workshops, 3rd ed., CMU/SEI-2002-TR-019, ADA405790 (Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2002).
21. F. Losavio, L. Chirinos, N. Lévy, and A. Ramdane-Cherif, Quality characteristics for software architecture, *J. Object Technology* **2**(2) (2003) 133–150.
22. IEEE, Std. 830-1998: IEEE Recommended Practice for Software Requirements Specification (1998).
23. G. Ruhe, A. Eberlein, and D. Pfahl, Trade-off analysis for requirements selection, *Int. J. Software Eng. and Knowledge Eng.* **13**(4) (2003) 345–366.
24. P. Zave and M. Jackson, Conjunction as composition, *Trans. on Software Eng. and Methodology* **2**(4) (1993) 379–411.
25. A. van Lamsweerde, R. Darimont, and E. Letier, Managing conflicts in goal-driven requirements engineering, *Trans. in Software Eng.* **24**(11) 908–926.
26. B. Nuseibeh, J. Kramer, and A. Finkelstein, A framework for expressing the relationships between multiple views in requirements specification, *Trans. Software Eng.* **20**(10) (1994) 760–773.
27. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, N degrees of separation: Multi-dimensional separation of concerns, in *Proc. Int. Conf. on Software Eng.* (1999), 107–119.
28. P. Gruenbacher, A. Egyed, and N. Medvidovic, Dimensions of concerns in requirements negotiation and architecture modeling, in *Proc. Int. Conf. on Software Engineering ICSE* (2000), http://www.research.ibm.com/hyperspace/workshops/icse2000/papers-index.htm (last visited: April 2006).
29. P. Grünbacher, M. Halling, S. Biffl, H. Kitapci, and B. W. Boehm, Repeatable quality assurance techniques for requirements negotiations, in *Proc. 36th Int. Conf. on System Sciences* (2003), pp. 9–17.
30. Å. G. Dahlstedt and A. Persson, Requirements interdependencies — Moulding the state of research into a research agenda, in *Proc. REFSQ — Workshop on Requirements Engineering for Software Quality* (2003), pp. 71–80.
31. B. W. Boehm, P. Bose, E. Horowitz, and M. J. Lee, Software requirements negotiation and renegotiation aids: A theory-W based spiral approach, in *Proc. 17th Int. Conf. on Software Engineering ICSE* (1995), pp. 243–253.
32. J. Park, D. Port, B. Boehm, and H. In, Supporting distributed collaborative prioritization for WinWin requirements capture and negotiations, in *Proc. Int. 3rd World Multiconf. on Systemics, Cybernetics and Informatics* (*SCI '99*), Vol. 2 (1999), pp. 578–584.
33. H. In and S. Roy, Visualization issues for software requirements negotiation, *Computer Software and Applications Conference COMPSAC 2001* (2001), pp. 10–15.
34. H. In, D. Olson, and T. Rodgers, A requirements negotiation model based on multi-criteria analysis, in *Proc. 5th Int. Symp. on Requirements Engineering* (2001), pp. 312–313.
35. J. Yen and W. A. Tiao, A systematic tradeoff analysis for conflicting imprecise requirements, in *Proc. 3rd IEEE Int. Symp. on Requirements Engineering* (*RE '97*) (1997), pp. 87–97.
36. P. Grünbacher, A. Egyed, and N. Medvidovic, Reconciling software requirements and architectures: The CBSP approach, in *Proc. 5th Int. Symposium on RE* (2001), p. 202.
37. P. Kruchten, The 4 + 1 view model of software architecture, *IEEE Software* **12**(6) (1995) 42–50.

38. J. Bosch, On the design of system family architectures, in *Proc. ICT-Architecture'99* (1999), http://www.serc.nl/lac/LAC-2001/lac-1999/docs/jan_bosch.pdf.
39. R. J. A. Buhr, Use case maps as architectural entities for complex systems, *Trans. Software Eng.* **24**(12) (1998) 1131–1155.
40. L. Xu, H. Ziv, D. Richardson, and Z. Liu, Towards modeling non-functional requirements in software architecture, in *Proc. Aspect-Oriented Requirements Engineering and Architecture Design Workshop* (*Early Aspects 2005*) (2005) http://trese.cs.utwente.nl/early-aspects-AOSD2005/workshop_papers.htm.
41. X. Franch and P. Botella, Putting non-functional requirements into software architecture, in *Proc. 9th Int. Workshop on Software Specification and Design* (1998), pp. 60–67.
42. F. Gross and E. Yu, Evolving system architecture to meet changing business goals: An agent and goal-oriented approach, in *Proc. From Software Requirements to Architectures Workshop STRAW* (2001).
43. R. Kazman, J. Asundi, and M. Klein, Quantifying the cost and benefits of architectural decisions, in *Proc. Int. Conf. Software Engineering* (2001), pp. 297–306.
44. H. In, R. Kazman, and D. Olson, From requirements negotiation to software architectural decisions, in *Proc. From Software Requirements to Architectures Workshop STRAW* (2001).
45. P. Clements, L. Bass, R. Kazman, and G. Abowd, Predicting software quality by architectural-level evaluation, in *Proc. 5th Int. Conf. on Software Quality* (1995).
46. R. Kazman, G. Abowd, L. Bass, and P. Clements, Scenario-based analysis of software architecture, *IEEE Software* **13**(6) (1996) 47–55.
47. A. Egyed and P. Grünbacher, Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help, *IEEE Software* **12**(6) (2004) 50–58.
48. A. Egyed, A scenario-driven approach to trace dependency analysis, *Trans. Software Eng.* **29**(2) (2003) 116–132.
49. A. Sutcliffe and S. Minocha, Scenario-based analysis of non-functional requirements, in *Proc. REFSQ — Workshop on Requirements Engineering for Software Quality* (1998), pp. 219–234.
50. A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech (eds.), *Rationale Management in Software Engineering* (Springer, Berlin, 2006).
51. L. Chung, D. Gross, and E. Yu, Architectural design to meet stakeholder requirements, in *Proc. 1st Working Conf. on Software Architecture WICSA* (1999), pp. 545–564.
52. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering* (Kluwer Academic Publishers, 2000).
53. N. Xie, N. R. Mead, P. Chen, M. Dean, L. Lopez, D. Ojoko-Adams, and H. Osman, *SQUARE Project: Cost/Benefit Analysis Framework for Information Security Improvement Projects in Small Companies*, Technical Note CMU/SEI-2004-TN-045 (Software Engineering Institute, Carnegie Mellon University, 2004).
54. B. Paech and K. Kohler, Task-driven requirements in object-oriented development, in *Perspectives on Requirements Engineering*, eds. J. Leite and J. Doorn (Kluwer Academic Publishers, 2003).
55. A. Herrmann and B. Paech, *Software Quality by Misuse Analysis*, Technical Report SWEHD-TR-2005-01 (University of Heidelberg, 2005) http://www-swe.informatik.uni-heidelberg.de/research/publications/reports.htm.
56. A. Herrmann and B. Paech, Quality misuse, in *Proc. REFSQ — Workshop on Requirements Engineering for Software Quality* (2005), pp. 193–199.

57. S. Easterbrook and B. Nuseibeh, Using viewpoints for inconsistency management, *Softw. Eng. J.* **11**(1) (1996) 31–43.
58. O. Laitenberger and J.-M. DeBaud, An encompassing life-cycle centric survey of software inspection, *J. Systems and Software* **30**(1) (2000) 5–31.
59. T. R. Browning, Applying the design structure matrix to system decomposition and integration problems: A review and new directions, *Trans. Eng. Man.* **48**(3) (2001) 292–306.
60. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (Addison-Wesley Longman, Reading, 1998).
61. L. Bass, P. Clements, R. L. Nord, and J. Stafford, Capturing and using rationale for a software architecture, in *Rationale Management in Software Engineering*, eds. A. H. Dutoit, R. McCall, I. Mistrík, and B. Paech (Springer, Berlin, 2006).
62. A. Herrmann and B. Paech, Benefit estimation of requirements based on a utility function, in *Proc. REFSQ — Workshop on Requirements Engineering for Software Quality* (2006).
63. International Standards Organization, ISO: Risk management — Vocabulary — Guidelines for use in standards, *ISO Guide 73* (International Standards Organization, Geneva, 2002).
64. N. Lassing, D. Rijsenbrij, and H. van Vliet, On software architecture analysis of flexibility, complexity of changes: Size isn't everything, in *Proc. Second Nordic Software Architecture Workshop* (*NOSA '99*) (1999), pp. 1103–1581.