1 Rationale Management in Software Engineering: Concepts and Techniques

A.H. Dutoit, R. McCall, I. Mistrík, B. Paech

Abstract: Rationale is the justification behind decisions. It is captured and used in many different forms during software engineering. While it has not achieved widespread use in practice, several approaches have emerged and successfully been used in selected projects. The goal of this chapter is to review the current state-of-the art of rationale management approaches and tool support in software engineering, and map future research directions.

Keywords: design rationale, rationale management, software engineering, software architecting, software requirements

1.1 Introduction

Rationale[1] is the justification behind decisions. It is captured and used in many different forms during software engineering (SE). The availability of rationale increases the developers' understanding of the system, making it easier to adapt or maintain. Being able to explain past decisions also facilitates the training of new members in a development team. However, rationale is often only captured partially and informally, often as natural language in design documents and in communication artifacts, making it difficult to access and maintain.

In the 1980s, the SE community, along with several others, started using rationale approaches. Process-based approaches, such as the use of Issue Based Information System (IBIS) described by Conklin and Burgess-Yakemovic [9], represent rationale as decision-making steps, capturing the argumentation behind designs as it occurs. Structural approaches, such as Questions, Options, and Criteria (QOC) [38], represent rationale as a space of alternatives and evaluation criteria, reconstructing rationale after decisions are made. In both cases, capturing rationale entails the elicitation

[1] Historically, much reasearch about rationale focuses on design and, hence, the term design rationale is most often used in the literature. In Sects. 1.1–1.5, which cover fundamentals of rationale management, we use the term *design rationale*. However, in Sects. 1.6–1.8, we use the term *software engineering rationale* to emphasize that rationale models are used during all activities of development, including requirements engineering, architectural design, implementation, testing, and system deployment.

and formalization of tacit knowledge, potentially introducing much overhead and disruption in the development process [4]. Rationale also features many elements and interdependencies, making it often difficult to keep up to date.

A rationale management system (RMS) is one that aims to address the above-stated issues. RMSs enable the capturing and accessing of rationale. The potential benefits of employing the services of an RMS include the following:

- Providing greater support to project management
- Improving dependency management
- Providing greater design support
- Helping support collaboration
- Supporting downstream users of design
- Allowing more detailed documentation
- Helping in requirements engineering
- Aiding in design reuse and ultimately provide a learning tool for evaluating design [36]

The complete rationale for even a small system is impossible to represent; consequently, developers are faced with selecting which rationale to represent in an RMS. Other implementation issues raised by researchers [46] are the formality (or informality) of the design rationale (DR) representation [21], and the approach to capturing rationale (e.g., reconstruction, apprentice shadowing, automatic generation). An RMS also aims to address the disruption caused by capturing rationale, recording rationale as a side effect of other activities, such as requirements elaboration, risk management, or process enactment.

The goal of this chapter is to review the current state-of-the art of rationale management approaches and tool support in SE, and map future research directions. Section 1.2 defines DR concepts. Section 1.3 discusses the fundamental DR approaches, such as IBIS, DRL, and QOC, from a historical perspective. Section 1.4 identifies and categorizes uses of design rationale. Section 1.5 identifies inherent limitations of DR approaches and proposes possible remedies. Section 1.6 discusses rationale in the specific context of SE, in terms of opportunities and a survey of the current state of the art. In Sect. 1.7 we synthesize our observations of Sect. 1.6 and propose an architectural framework for RMSs in SE. We conclude and discuss future research directions in Sect. 1.8.

1.2 Design Rationale Fundamentals

Systematic documentation of rationale for practical decisions began more than 35 years ago with work on rationale for design [30], in particular, design of buildings and cities. In the 1980s, interest in rationale spread to other fields involved with design, including SE and mechanical engineering. In recent years, researchers in SE have begun to look at rationale for activities other than design, and we argue later that this is an essential trend for the future of the field. Nevertheless, rationale for design remains the dominant theme in rationale research in SE. To understand the history and current state of the field, it is essential to understand the work done on design rationale. This section therefore defines the term *design rationale* and introduces fundamental characteristics of DR approaches.

1.2.1 Definitions

Our definitions are meant to accommodate many points of view about DR. The definitions we have chosen are similar to those given by MacLean et al. [38]:

1. We start by defining a *design process* as one that aims at devising an appropriate *design* for an artifact. A *design* we define as an artifact description that is detailed enough for use in implementing (constructing) that artifact. We consider a design *appropriate* if the artifact described would satisfy requirements while not being unacceptable in other ways, e.g., by producing an unacceptable set of side- and after-effects. Two major categories of *artifacts* are (1) physical artifacts, such as buildings, cities and computer hardware, and (2) cognitive artifacts, such as notation systems and software.
2. We define a *designer* to be anyone participating in a design process. This definition depends on how the term *participating* is defined and leaves open the possibility that users and clients could be designers.
3. *Design rationale (DR)* is the reasoning that goes into determining the design of the artifact. It can include not only direct discussion of artifact properties but also any other reasoning influencing design of the artifact. Note that our definitions do not imply that design starts only after requirements have been fully determined. During requirements specification many design decisions are made and these are relevant for design rationale. Similarly, design does not stop before implementation begins. Feedback from implementation and testing could be part of the design rationale.

1.2.2 Making Sense of the Varieties of DR Approaches

There are already many different approaches to DR, and more are coming into existence on a regular basis. This multiplicity of approaches shows that the DR field is healthy, but it also creates the need to make sense of this variety by finding organizing principles. In this section, we will therefore look at some ways of characterizing DR approaches to facilitate comparison, reveal trends and highlight issues. Describing even briefly the many approaches used is beyond the scope of this chapter, but we will describe some approaches that are frequently used and others that challenge widely held assumptions.

There are three ways of characterizing approaches to DR that reveal fundamental differences and similarities among them. One is to look at the way in which DR is *represented* and *processed* in an approach. Another is to describe the extent to which approaches are *descriptive or prescriptive* with respect to design. The third is to describe their *intrusiveness* in the design process.

Representation and Process Implementation

It is useful to characterize DR approaches by how they represent rationale and by how they implement basic DR processes.

- *DR representation form***.** Almost invariably, DR is represented by being divided up into *chunks* that are assigned certain properties and/or relationships. By far, the most common way of doing this is through use of a DR schema, i.e., a fixed, semi-formal, conceptual schema that represents the types of elements (chunks), properties and relationships in terms of which DR is represented. An alternative approach to DR representation involves linking DR chunks to features of the artifact they discuss. Yet another approach is to link DR chunks to steps in a description of the process of using the artifact.
- *DR process implementation***.** Using a DR approach involves making commitments about how to implement three basic processes:
  - Capturing rationale, the process of eliciting rationale from designers and recording it
  - Formalizing rationale, the process of transforming rationale into the desired representation form, such as a DR schema
  - Providing access to rationale, the process of getting recorded rationale to the people who need it

A given rationale approach typically indicates how each of these processes is to be implemented. It indicates which entities perform processes, i.e.,

whether they are done by computers or humans, and, if humans, which by which humans. It also indicates when the processes are carried out, e.g., during design or afterwards.

Capturing rationale might be done in different ways. Designers might do it themselves or have it done by nondesigners who are specialists in DR documentation. A third possibility is to extract DR from records of communication among participants in a project. A fourth is to capture it as a side-effect of the use of design-support software.

Traditionally, capturing and formalizing rationale were combined in a single operation. In recent years, however, alternative approaches separate the formalizing of rationale from its capture. One way of implementing formalization is to have it done by the same people who state the rationale. An alternative is to have it formalized by personnel specially trained in formalizing rationale. Yet another approach is to use software tools that partially or completely formalize informally stated rationale.

The most common approach to accessing DR is through use of a system that lets users browse a hyperdocument containing the rationale. Conventional information retrieval (IR) search techniques can also be used. A third approach to accessing DR uses knowledge-based critics that alert users to the existence of DR they might need.

Descriptive or Prescriptive

- *Descriptive approaches***.** Some approaches to DR are aimed only at describing whatever thinking processes designers might choose to use. Such approaches make no attempt to alter designers' reasoning. They might, however, use records of DR to improve processes outside of design, such as implementation, maintenance, or reuse of designed artifacts. They might also use DR to bring new members of a design team up-to-date. Such approaches are only interested in DR as a *descriptive model* of designers' thoughts, utterances or actions.
- *Prescriptive approaches***.** On the other hand, some approaches are aimed at *improving design processes* by improving the reasoning of designers. They typically attempt to remedy perceived deficiencies in design reasoning by making it more correct, more consistent and more thorough. As with descriptive approaches, prescriptive approaches can create records of DR that are used to improve processes outside of design. It should also be noted that the descriptive and prescriptive are not always mutually exclusive. For example, some approaches are primarily descriptive in intent yet also have some prescriptive goals.

Intrusiveness

Another useful way of characterizing DR approaches is by their intrusiveness in the design process. This includes not only how intrusive they are but in what respects they intrude. Thus, an approach might be highly nonintrusive during capture of DR but relatively intrusive during retrieval and display of rationale. Measures of intrusiveness can include the degree to which a DR approach dictates the way design is done as well as the amount of extra effort required to use the approach. And the acceptability of intrusiveness may differ for capture, formalization and access.

- *More-intrusive approaches*. Most proposed DR approaches are highly intrusive with respect to DR capture in that they intervene in the design process to guide the way rationale is elicited from designers. Typically such interventions use a DR schema that defines what types of elements of rationale should be elicited from designers and how these elements should be linked together.
- *Less-intrusive approaches*. Over the past 15 years, a number of researchers have sought less intrusive ways of capturing and formalizing DR. This is due to concern about difficulties experienced in getting rationale capture to work in design projects. Specifically, many of these researchers believe that intrusiveness has been a central obstacle to effective capture of rationale, though there have been few complaints about intrusiveness as a barrier to accessing it.

One might imagine that prescriptive approaches were generally intrusive while descriptive modes were nonintrusive, but the actual story is not so simple. For example, QOC [38] is a highly intrusive yet primarily descriptive, while use of domain-oriented issue bases in Procedural Hierarchy of Issues (PHI) [17] is prescriptive yet highly nonintrusive. It is true, however, that descriptive approaches sometimes facilitate use of nonintrusive means to capture DR. Examples include the capture of DR from CAD usage [47] and use of natural language processing to structure computer mediated communication in design [45].

By itself, no representation scheme, such as a DR schema, is intrusive. It only becomes intrusive when used with an intrusive processing implementation mode, as when a schema is used to guide rationale elicitation. In such cases, however, different schemas can have different levels of intrusiveness. Generally, a more fine-grained schema will be more intrusive, because it makes designers perform more categorization and linking tasks. In addition, schemas that organize rationale in a way that is different from the way designers would intuitively organize it create a

cognitive dissonance that adds to the cognitive overhead that designers must cope with.

1.3 Approaches to Design Rationale

This section focuses on three argumentative approaches to DR: IBIS, QOC, and Decision Representation Language (DRL). It thereby gives an introduction into the most prominent issues in providing DR support. It also contrasts argumentative approaches with problem-based, scenario-based, and generative approaches.

1.3.1 Three related approaches to argumentation – IBIS, QOC, and DRL

The most commonly used way of treating DR is as a type of *argumentation* that is structured according to a given schema. There are many ways in which DR argumentation might be structured, but there have historically been two major branches of thought. One branch uses some variant of the schema for argument structure devised by Toulmin [63]. The other uses one of a group of DR schemas having IBIS, QOC, and DRL as its most prominent members. Interest in the former branch seems to have faded over the past 15 years, while the latter continues as perhaps the dominant trend in the field. We will concentrate exclusively on the latter approach to argumentation. In particular, we will examine the similarities and differences among IBIS, QOC, and DRL.

IBIS

Historically, the DR movement began with Rittel's IBIS (Issue-Based Information System), which was not a software system but a way of modeling argumentation [30]. By 1967, Rittel had become convinced that design problems were *wicked problems* and fundamentally different from the well-defined problems of science [7, 54]. He called for an "argumentative approach" to wicked problems and used IBIS to implement this approach [55]. In the 1970s and 1980s he applied IBIS to large-scale projects in planning and policy making for the United Nations, the Commission of European Communities and the West German government. Other researchers applied IBIS to architecture and planning [41].

In the mid-1980s Conklin discovered Rittel's writings on *wicked problems* and saw this theory as a way of understanding the profound difficulties

that software design had run into. He then contacted Rittel, who told him about IBIS [53]. Conklin then adapted IBIS for use in SE and created the graphical IBIS (gIBIS) hypertext system to support this use of IBIS [8, 9].

Rittel's IBIS had the following elements:

- Issues
- Positions
- Arguments
- Resolutions

In addition, there was a variety of inter-element relationships.

IBIS considers the pros and cons of *positions*, which are proposed alternative answers to questions, which are called *issues*. Positions are evaluated on the basis of *arguments* about the relative merits of the positions as well as the merits of other arguments. In principle, these arguments can range in size from a brief sentence to many paragraphs, though typically they are one to three sentences in length. Issue discussion often involves a multilevel structure of arguments for and against positions as well as arguments for and against other arguments. The decision on which position to accept is called the *resolution* of the issue.

Rittel's IBIS used several relationships to link different issue discussions. These included *more general than*, *logical successor to, temporal successor to*, *replaces* and *similar to*. Variants of IBIS developed by others often used relationships that differed from Rittel's to lesser or greater degrees.

Rittel looked at IBIS as a way of representing debate of controversial questions that arise in design. In fact, he intended IBIS to be a means for promoting debate of such questions from many different points of view. He was much less interested in the treatment of noncontroversial design questions, which were labeled *trivial issues* and not dealt with by IBIS.

Rittel's approach was from the outset both prescriptive and intrusive, as were almost all of his IBIS projects. Other researchers, however, have sought much less intrusive ways of using IBIS.

PHI

Procedural hierarchy of issues (PHI) [40, 41] extended IBIS to noncontroversial issues and rethought the relationships between issues. The centerpiece of PHI is the *subissue* relationship, where one issue's resolution depends on the resolution of another. In PHI a design project is a quasi-hierarchical structure of subissues that resembles a *calling structure* of subroutines in procedural programming. (A quasi-hierarchy is a directed acyclic graph with some added cyclical structures.) This is in contrast to the "spaghetti" structure of issue networks in Rittel's original version of

IBIS. PHI was intended to facilitate the creation of larger and more comprehensive models of design reasoning. While Rittel's IBIS typically dealt with 30–50 issues in a project, PHI typically dealt with 200–400 issues.

PHI also revised IBIS to better reflect actual practice in the IBIS community in the 1970s. The term *answer* was adopted (instead of *position)*, since this term was widely used in this community. Also, the concept of *subanswer* was added so that hierarchies of answers could be represented. Such hierarchies were also in widespread use by IBIS practitioners, yet had no formal status in IBIS. PHI extended this naming scheme to arguments: the arguments on other arguments got labeled *subarguments*. This meant that hierarchies of issues, answers, and arguments could all be dealt with using a uniform naming scheme.

Originally PHI was both prescriptive and intrusive. Over the past 20 years, however, PHI has been used in ways that are increasing nonintrusive. The central tenet of PHI was that the key to improving design reasoning is to *raise more subissues*. In other words, the attitude behind PHI was that better treatment of an issue means thinking about what other issues its resolution depends on. For example, a house designer might raise the issue, "How many stories should the house have?" You can do a better job resolving this issue if you consider what other issues (*subissues*) the resolution depends on. For example, the number of stories for a house might depend on the following subissues:

- How much land is available for the house?
- How many people will live in the house?
- Will elderly or disabled people be living in the house? (since such people may have difficulty with stairs)

A number of hypertext systems were created to support PHI, starting with PROTOCOL [41] in the late 1970s. This was succeeded by MIKROPLIS [39, 44] in the early 1980s, which in turn evolved into PHIDIAS [42, 43] in the 1990s. In this period, the JANUS [17] system was also created to support delivery of PHI-based rationale to designers.

A crucial application of PHI is to create *domain-oriented issue bases*. These are structured collections of issues, answers, and arguments that have a high degree of recurrence in different projects in a given problem domain, e.g., design of houses. An issue's rationale might not include its resolution, since this varies from project to project. There is no claim that an issue base contains all the rationale for a project; instead, it is merely a convenient starting point for creating that rationale. Typically, much more work goes into designing a domain-oriented issue base than it is reasonable to spend on DR design in a single project. This extra work pays off when an issue base is used to inform many design projects within a domain.

QOC

A second schema for argumentative DR is used by the QOC approach to DR. While *QOC* stands for Questions, Options, and Criteria, it actually has six major types of elements:

- Questions
- Options
- Criteria
- Assessments
- Arguments
- Decisions

In addition, QOC has relationships between elements, including inter-question relationships. Since QOC's schema appears similar to IBIS's, it will be useful to point out the differences between the two as we explain QOC. Like IBIS, QOC centers DR on the discussion of questions. A crucial difference between QOC and IBIS is that while IBIS's questions, i.e., *issues*, can concern any design topic, *QOC's questions deal exclusively with features of the artifact being designed.* An example of a QOC question given by MacLean et al. is, "How should the scrollbar be displayed?" This, of course, would also count as an issue in IBIS. While QOC potentially deals only with a subset of the questions that an IBIS might deal with, QOC has the advantage of not allowing the designer to ignore questions about features of the artifact. IBIS, by contrast, does not mandate that such questions be dealt with, though they typically are.

Alternative answers to QOC questions are called *options*. Options represent possible features of the artifact being designed. These are identical to *positions* on IBIS issues that deal with the features of the artifact. The following are examples of options adapted from an account by MacLean et al. [38]:

1. Have the scroll bar permanently fixed to the edge of the window
2. Have the scroll bar invisible normally but visible when the cursor 'rolls over' the edge of the window

Questions and their options in QOC together constitute the design space, which corresponds to the set of possible alternative designs for the artifact. The use of QOC is referred to as design space analysis.

One crucial respect in which QOC differs from IBIS is in the way in which the alternative answers to questions are evaluated. QOC, first of all, requires use of explicitly stated *criteria* to evaluate proposed answers (*options*). Criteria indicate desirable properties of options or requirements they should satisfy. Second, QOC requires that answers be linked by positive or negative links to criteria, a positive link indicating that an option does well according to a criterion and a negative link indicating that it does

poorly. The links of criteria to options are called *assessments.* IBIS does not require this sort of consistent evaluation. It only asks for arguments for or against the answers (*positions*). Nevertheless, each assessment in QOC could be represented in IBIS as an argument for or against a position.

Like IBIS, QOC can have *arguments* that challenge or support any element. In QOC emphasis is given to arguments on assessments. As with IBIS, argumentative structures can have multiple levels of arguments on arguments. Finally, QOC has *decisions* indicating which options to accept for each question. These correspond to *resolutions* of issues in IBIS.

In summary, there are two main things that distinguish QOC's schema from IBIS's. One is that QOC's questions always have possible answers (*options*) that describe properties of the artifact being designed, whereas *issues* in IBIS can include these questions as well as the many other questions that arise in a design project. QOC has no way of dealing with the multilevel *subissue* structures that are the hallmark of the PHI version of IBIS. On the other hand, the IBIS schema cannot guarantee that QOC-type questions are addressed. The second thing distinguishing the schemas is that QOC uses *assessments* indicating how answers (*options*) perform with respect to explicit *criteria*. While these assessments can be stated in IBIS as arguments, IBIS has no explicit representation of criteria as elements.

The goals of QOC approach are primarily descriptive, in that the main purpose of the system is to create a description of designers' rationale that is sufficiently detailed to inform other phases of the artifact lifecycle. QOC's *process implementation mode* is intrusive in its use of designers' time to guarantee that the description is thorough.

The authors of QOC have not created software to support QOC, though a number of other researchers have incorporated such support into their systems.

DRL

Decision Representation Language (DRL) [34] began as an extension of the Potts and Bruns model of DR [49], which was itself an extension of IBIS. Lee and Lai [35] argue that DRL is more expressive than other argumentation schemas in the sense that it enables the answering of a broader range of questions that might arise in various phases of the artifact lifecycle. What this claim primarily boils down to is having DRL provide a finer level of granularity in certain parts of its schema. Lee and Lai do not claim that DRL provides more comprehensive coverage of DR than other approaches. They state that their schema is for *decision rationale* and does

not deal with all aspects of *design* rationale, such as deliberations on how to generate design alternatives. IBIS can deal with these aspects.

The primary elements of DRL are as follows:

- Decision problems
- Alternatives
- Goals
- Claims
- Groups

DRL also has various relationships between these elements. Many of these elements and relationships correspond to the aspects of QOC and IBIS.

A *decision problem* is something to be decided. Lee and Lai claim that a decision problem is equivalent to both a question in QOC and an issue in IBIS, but they appear not to recognize that a QOC question is not equivalent to an IBIS issue. It seems, however, that a decision problem actually corresponds to a QOC question rather than an IBIS issue because *alternatives,* i.e., the alternative solutions to decision problems, are always artifact features. An *alternative* is the same as an *option* in QOC and can be represented as a *position* in IBIS. A *goal* is used for comparative evaluation of alternatives and corresponds to a *criterion* in QOC, but has no explicit representation in IBIS. Alternatives can be linked to goals by *achieves* relationships, which correspond to *positive assessments* in QOC. Alternatives are evaluated by *claims* about the *achieves* relationships between alternatives and goals, a scheme that mirrors QOC's use of arguments to discuss its *assessment* relationships between *options* and *criteria*. These can be modeled as IBIS arguments, but doing so buries the *goals* in texts rather than explicitly representing them as elements. Further claims can be linked to other claims by *support* or *deny* relationships, which are semantically identical to relationships in both QOC and IBIS.

So far DRL looks nearly identical to QOC, but DRL has some features not found in QOC or IBIS. One is a *presupposes* relationship between claims. In addition, each claim has three attributes: *evaluation*, *plausibility*, and *degree*, the value of the evaluation attribute being determined from the values of the plausibility and degree attributes. *Plausibility* represents the likelihood that the claim is true, and *degree* represents the degree to which it is true. DRL also allows the creation of *goal–subgoal* hierarchies. DRL also includes a *subdecision* relationship between decision problems that corresponds to a *subissue* relationship among issues in PHI. Also DRL's *claims* represent a sentential level of granularity for argumentation, whereas IBIS *arguments* provide only a syllogistic level of granularity.

The stated goal of making DRL more expressive than other methods suggests that the system is primarily descriptive, but a number of the questions that Lee and Lai list in defining DRL's expressiveness have implications for improving design. So DRL appears to be more prescriptive than QOC, though less prescriptive than IBIS.

Lee created SYBIL, a knowledge-based hypertext system to support collaborative use of DRL [32, 33]. SYBIL is built on ObjectLens [31], a general tool for building CSCW applications.

IBIS, QOC, and DRL Compared

DRL's schema seems to correspond to a superset of QOC's, because every QOC feature appears to correspond to a DRL feature, though not the other way around. Both QOC and DRL are more expressive than IBIS in that they provide more fine-grained models of the argumentation that directly deals with evaluation of artifact features. But IBIS is more comprehensive in that it can represent the discussions of some design questions (*issues*) that neither QOC nor DRL treats. Lee and Lai state that DRL deals only with *decision rationale* and that this does not include all of *DR*. Since DRL is a superset of QOC, this limitation would also apply to QOC. Neither Rittel's IBIS nor its PHI variant has this limitation.

What aspects of DR are left out of *decision rationale*? Lee and Lai give only one example: discussions related to generating feature alternatives. In PHI there are two major classes of subissues: those that help in *evaluating* alternative answers (*positions)* and those that help in *generating* them. Lee and Lai are, in effect, saying that the latter cannot be represented in DRL, and, by implication, QOC. A simple example of an alternative-generating subissue might be, "How have multiple 'screens' of information been displayed in other software systems?" Such an issue identifies possible alternatives for artifact features, such as, "by scrolling" and "by showing multiple 'cards' of information, as in NoteCards and HyperCard." Yet such an issue it is not a *decision problem*, because answering the issue *does not decide which feature alternative to adopt*. Neither DRL nor QOC has any explicit way of dealing with such issues.

QOC, however, can deal with certain design questions that do not have alternative answers that are artifact features, for some criteria in QOC are represented as questions. Such questions would clearly count as issues in IBIS (and subissues in PHI); but criteria in QOC (goals in DRL) only deal with things, such as requirements, that can be used to directly evaluate alternative artifact features. One example of questions that neither count as QOC *questions* or *criteria* is found in an example of an area of design discussion where MacLean et al. acknowledge that the "overlap" with

Design Space Analysis "is relatively weak." In this example from an empirical study software designers say the following:

> What is causing the long queue[?] Is it people just going through these steps, or is it people adding options to other services, and then using the other option?

MacLean et al. describe the process of addressing such questions as building an ad hoc *theory*, something that QOC does not handle. IBIS requires no special new way of handling such questions; they are simply *issues*.

Originally, both the IBIS and QOC schemas were used in intrusive DR approaches. But over the past 15 years, research on the PHI has sought to devise nonintrusive means for capture, formalization and delivery.

The QOC approach concentrates on the *design of rationale* rather than *recording the processes of rationale generation*, such as in *the process implementation mode* in the use of IBIS described by Conklin and Burgess-Yakemovic [9]. Apparently, on the basis of this difference MacLean et al. have claimed IBIS is restricted to capturing rationale "on the fly" and therefore only records a history of a design process, whereas QOC records the "logical argumentation." Actually, over its 35-year history IBIS has often been used in the same way QOC is used, i.e., to create "logical argumentation." For example, using PHI to create domain-oriented issue bases [17] is entirely concerned with designing "logical" rationale and leaves no record of the process by which the rationale is produced. In other words, the schemas of Rittel's IBIS and its PHI variant have been used with different DR *process implementation modes*.

Here, it is important to point out a fact that makes it difficult to compare DRL with QOC: writings on DRL generally contain little information about *process implementation mode*. DRL seems not so much a DR approach as a schema that might be used in various DR approaches.

It is perhaps surprising that there are so few significant differences in the schemas of IBIS, QOC, and DRL. The differences that do exist appear to be features of one schema that could profitably be added to the other two. In fact, MacLean et al. state that they would like to make QOC more like DRL, and Lee and Lai say that they see DRL as extending the changes that PHI made in IBIS. This suggests that it might be both possible and useful to combine the three schemas.

1.3.2 Approaches to DR that Go Beyond Argumentation

Problem-Based Evaluation

Lewis et al. [37] present a novel approach for evaluating alternative features of an artifact. They describe their own software design process as using a suite of problems for conceptual evaluation of different proposals for a computational environment they devised. Their experiences may sound familiar to other software designers, and yet no other DR approach has taken such experiences into account. Among other things, their work suggests that argumentation alone may not be the only, or even the best, means of evaluating alternatives, and this, in turn, challenges the sufficiency of existing argumentative approaches to DR. Implications of the Lewis–Riemann–Bell insight for other types of design, including other types of software design, need to be looked into. How their work might augment an argumentative approach to DR also needs to be worked out.

Scenario-Based Evaluation

Carroll and Rosson [6] propose a way of evaluating software features that does not document the reasoning of designers but rather the potential reasoning of users in hypothetical scenarios of human–computer interaction. While this is fundamentally different from standard argumentative approaches, a potential point of connection with argumentative DR is that the four examples of scenarios that Carroll and Rosson provide are all *question-answering processes*. Another connection is that scenario-based design involves the analysis of *claims*. Carroll and Rosson emphasize, however, that the claims they study deal only with the psychological consequences of artifact features and are "embodied" in, and thus inferable from, the artifact and its use. They see their work as a more abstract version of the problem-based approach of Lewis, Riemann, and Bell. They also see it as similar to QOC in some ways, but as being at a higher level of analysis and more connected to use situations.

Generating DR from Data and Models

Gruber and Russell [23] argue that argumentative schemas do not include all the rationale that designers use, because all of them are prescriptive about what information is relevant. No collection of DR, they claim, could answer all of the questions that might be raised about the rationale for an artifact. Rather than having designers elicit highly detailed models of their rationale, it would be better to collect engineering data and models and then later use these to infer DR in response to questions that arise about it.

1.4 Uses of DR and DR Methods

There are many potential uses of DR, some aimed at improving design, others at improving other phases of the artifact life cycle. Frequently proposed uses are listed below. Note that there are some overlaps and dependencies among items in this list. We group them into four main categories: the first focuses on collaboration, the second on reuse and change, the third on quality improvement, and the fourth on knowledge transfer.

1.4.1 Supporting Collaboration

Promoting Coordination in Design Teams

DR can help to coordinate many aspects of a design team's work. Different members of a team can use a common repository of DR to understand what others in the team are doing and what the consequences are for their own work. This can promote the identification of both potential conflicts between team members and opportunities for mutual support.

Exposing Differing Points of View

One use of DR is to expose differing points of view. Sometimes these are merely differences of opinion on detailed issues, but sometimes they are also profound differences of worldview on fundamental topics, e.g., open-source vs. de facto commercial standards. Sometimes they arise from differences in domain expertise in a functionally differentiated design team. Sometimes they arise from the different goals of different stakeholders in a project. Exposing differing points of view and the reasoning behind them was a central goal in Rittel's use of IBIS. Not all DR approaches share Rittel's aim of promoting debate. Some are more aimed at promoting a rapid convergence on agreement.

Facilitating Participation and Collaboration in Design

DR can be used to promote both collaborative and participatory design. Rittel argued that participation by users in design is often inhibited because they do not understand what rationale designers are using, what questions they are addressing, what alternative answers they are considering, what arguments they are using. He looked at IBIS as a way of making designers' reasoning transparent, i.e., a glass box rather than a black box, and thus empowering users to ask questions and to make comments and suggestions.

Similarly, he saw collaboration as also being inhibited when members of a design team did not understand the rationale being used by other members. DR approaches other than IBIS can also be used this way.

Building Consensus

Many users of IBIS have complained that it lacks adequate means for promoting consensus and reaching decisions. Other DR methods might be better for creating consensus for the simple reason that they do not go to such lengths as IBIS goes to promote debate.

1.4.2 Supporting Reuse and Change

Supporting Future Changes

The most commonly mentioned reason for using DR in SE is to support future changes in software, a problem that is perhaps more pressing in this field than in any other design or engineering field. This does not necessarily require a prescriptive approach to DR. This goal might be well served by approaches that merely record what designers happened to think. People who want to make future changes need to understand the effects of those changes; knowledge of the rationale for the design can help in achieving that understanding. Sometimes that rationale may also reveal that some planned changes are actually inappropriate. It is not uncommon that a design feature that seems wrong to a new designer was originally arrived at, through a painful process of trial-and-error in which all the "intuitive" approaches failed. Without a record of the rationale, this painful process might have to be repeated, perhaps many times.

Supporting Reuse

Software reuse is often considered the "holy grail" of software design. But before software can be reused it needs to be understood and/or modified. This requires knowing the reasoning behind its original design. DR can also help to identify parts of software that might be extracted and reused.

1.4.3 Improving Quality

Increasing Consistency of Decisions

Often it is only by making rationale explicit that consistency can be achieved. For example, it is not uncommon in large projects for the same

decision tasks to be done by different groups within the design team. Recording rationale makes it easier to identify this fact and to make sure that decisions are mutually consistent. This use of DR is prescriptive, for it seeks to change the way designers think, i.e., making it more consistent. Though this use of DR requires methods that go beyond mere historical description of designer's rationale, such description may be of value because it exposes designers' reasoning to critical scrutiny.

Verifying Designs (Supporting Traceability)

This use of DR requires an explicit linking of requirements criteria to the descriptions of artifact features that satisfy these criteria. In the case of software, it also suggests the desirability of linking the criteria to actual features of the implemented software. This requires a schema that makes criteria explicit, as QOC and DRL do, rather than schemas where criteria are embedded in larger arguments, as is the case with IBIS.

Supporting Maintenance

One possible use of DR is to support debugging, fixing problems, and extending the functionality of an artifact. This problem is probably more critical with software than with any other type of artifact. DR can be used to spot conceptual errors in design as well as implementation errors, and errors of omission as well as errors of commission.

1.4.4 Supporting Knowledge Transfer

Learning from the Past

To learn from the past, we need to understand the reasoning behind past decisions. Most DR researchers maintain that this can best be done through explicit recording of the rationale for those decisions, something that requires nothing more than a descriptive model of whatever it was that the designers were thinking when they made decisions. Gruber and Russell [23], however, have presented evidence that designers are often able to effectively reconstruct the rationale for past designs from data other than an explicit record of rationale. These authors even suggest that it may be more useful to record such data rather than the rationale itself.

Validating Designs

To maximize learning from the past, we need to be able to compare designers' expectations about the consequences of their decisions with the actual consequences. This requires more than an understanding of the reasoning behind past decision; it requires evaluation of artifacts in use. One approach to doing this is found in case-based reasoning (CBR) projects by Kolodner [29]. Especially interesting is the ARCHIE project, which records the experiences of users of artifacts (buildings) and links these experiences to representations of the artifact.

Organizing and Delivering Reusable Knowledge

The issue of learning from the past is also fundamentally connected to the reuse of knowledge. Reuse can be thought of not only as using the successful ideas and rationale from the past, but also a matter of preserving records of what not to do. There is no point in reinventing the wheel, but it makes even less sense to reinvent the square wheel. Thus, the blunders of past designers represent an important type of reusable knowledge.

There are two basic approaches to the reuse of knowledge: *the case-based approach*, mentioned above, and what we might call *the generalized approach*. The latter term is intended here as an umbrella term for a number of approaches that try to put knowledge in a generalized form that goes beyond the mere annotation of individual cases. There are currently a number of generalized approaches, including patterns and issue bases.

Patterns, as used in SE, constitute one of the most heavily used approaches for organizing reusable knowledge [19]. Integrating rationale more completely into such patterns could be an important way of making rationale reusable. The patterns used in SE ultimately derive from Alexander's concept of *pattern* used in his work on architecture and urban planning [1]. This pattern concept has rationale explicitly built in, though this rationale is relatively unstructured.

Domain-oriented issue bases have only been created with PHI. Such issue bases contain hierarchies of issues, positions, and arguments that are commonly raised in projects in the domain. Most issues are left unresolved and designers are invited to make their own minds up on the issues. Wherever the software technology permits, issue bases are extensible by designers, who can add to them and even edit them for use in specific projects.

Supporting Training

One use of DR is to bring new members of a design team up-to-date on work in a current project. DR can function as a sort of larger-scale

version of an FAQ, so that a new member can understand the rationale for the current state of the artifact's design before suggesting changes to it.

Providing External Design Memory

DR is useful as a memory aid for members of a design team. This is especially important where projects go on for long periods of time and where designers leave the team. It is very important when designers leave a project that all knowledge of their project rationale does not leave with them.

1.5 Limitations of Current DR Approaches and Software

Despite the many approaches to DR suggested and the many software systems devised, DR has not found ongoing use in real-world design. There are cases where DR has been applied successfully; but these often depend on special circumstances, such as the presence of a "DR champion" [9], that cannot be expected to exist in the majority of cases.

There are a number of ways in which DR methods can fail to be used in practice. One is the use for eliciting and recording rationale from designers, which is generally known as *capturing DR*. The other way is for retrieval and display of recorded rationale, what we shall call *providing access to DR*. We will focus here on the former, because it has been the central obstacle to the practical applications of DR. In fact, so little DR has been captured to date that has been relatively little opportunity to investigate the problem of DR access in real-world settings.

1.5.1 The Capture Problem

There seems to be a broad consensus that DR capture has generally not worked in practice. Designers have typically resisted rationale capture. Why they resist is a central question in research on DR capture and one of the most important issues in the DR field. If we were certain of how to answer this question, we would know the conditions, if any, under what the capture problem is solvable and how to begin solving it.

There are a number of possible explanations for resistance to DR capture. Some researchers point to its intrusiveness as the problem. One kind of intrusiveness is due to the work required for capture. Most capture involves designers writing up their rationale in a given DR schema. This requires a great deal of work in addition to the normal work of design.

Other reasons for resistance to capture can include political and legal factors. Designers might not want their bosses or the public to know the real reasons for their decisions. They might also want to protect themselves from potential law suits. There is also the problem that any argument can be a double-edged sword that provides others with a way to attack decisions made.

For descriptive approaches, the extra work of DR capture can be a fundamental problem. Since such approaches do not aid design, those who record the rationale are unlikely to be the ones who use it. Designers might thus have little motivation to do the capture. Descriptive approaches run afoul of Grudin's principle that collaborative systems tend to fail when those who do the work are not the beneficiaries of that work [24, 25].

Grudin argues that in developing commercial off-the-shelf (COTS) software DR capture might not pay off at all for later phases. COTS projects are failure prone, because (1) most products fail commercially and (2) up to 90% of projects are not completed. Failed projects do not need DR, and using resources for its capture could make failure more likely.

Grudin also suggests that in COTS development design decision making is often highly distributed. Experts and stakeholders of many types shape the design. There is often no way of compelling these individuals to share their rationale, much less to use a DR software system.

Grudin analyzes DR capture in three additional development contexts. For *in-house development in organizations* and *competitively bid contract development* he finds that incentives for DR capture offset some of the disincentives he found in COTS development. For *customized software development*, however, the only real disincentive he finds is that the firms doing it are often small and lack resources to invest in new software tools.

Another possible explanation for resistance to DR capture is that the quantity of work required for capture is greater in time than designers have for in a project even if they want to do it. Design is an intense activity that tends to absorb all the resources of time and personnel available.

For prescriptive approaches, there is supposedly a benefit to designers for capturing DR, so designers should be more motivated to do it. Yet even here, it has not succeeded. A simple reason for this might be that investing resources in DR capture has less benefit than investing it in design.

Another possible reason for the failure of DR capture in both descriptive and prescriptive approaches is that DR capture might actually be detrimental to design in ways that go beyond its cost in resources. For example, Fischer et al. [17] use Schön's theory of Reflective Practice [56] to argue that DR can actually disrupt designers' thinking. Schön sees design as involving two very different cognitive processes: an intuitive process of skillful action, which he calls *knowing-in-action,* and a reasoned process of

reflection, which he calls *reflection-in-action*. He sees design as continually alternating between the two. The two processes cannot be done simultaneously, because reflection disrupts knowing-in-action. Reflection is only productive when intuition fails to cope with some new circumstance arising in design. To Fischer et al. this means that the explicit argumentation of DR is only appropriate for reflection-in-action. This in turn implies that rationale capture can actually degrade the quality of design if it is *intrusive into the intuitive processes of knowing-in-action*.

A more radical position on intrusiveness is taken by Shipman and Marshall [57]. They argue that semi-formal schemas are themselves the problem. As they see it, all such schemas are obstacles to capture information. They advocate doing away with structured user input and using only informal input.

Another possible explanation for the resistance to capture is that we still are not collecting the right information. The work of Gruber and Russell, Lewis, Riemann, and Bell as well as that of Carroll and Rosson suggest that argumentation by itself may not be enough to account for designers' reasoning. There are enough dissenters from the argumentative view of DR to leave room to doubt that we are capturing the right information. Nevertheless, there is little evidence to date that differences in information recorded have made any difference to the success of DR capture in practice.

1.5.2 Approaches to Solving the Capture Problem

Traditionally, DR literature has emphasized that devising the right schema, i.e., one that captures the right information and structures it correctly, is the way to solve the problems of DR usage. Yet designers' resistance to DR capture exists regardless of what schema is used. Solving the capture problem will require research on more than schema design.

One direction taken by researchers working on solving the capture problem is to try to reduce the intrusiveness of DR capture, either by reducing the work of DR capture or reducing its disruptiveness in design or both. The MIKROPLIS [39, 44] and gIBIS [8] hypertext systems reduced the work of managing DR by providing extensive support for browsing, modification, and retrieval. This, by itself, however, was not enough. The cognitive overhead of DR capture remained daunting.

One approach to reducing the cognitive overhead of capture is to use the strategy of *differential description*, in which designers only need to describe how the rationale for the current project differs from other rationale. One way to do this uses domain-oriented issue bases in PHI [18]. These contain rationale commonly used in projects in a given domain, including

commonly raised issues, positions and arguments. Designers need to add only the missing information, including their decisions on the issues.

There are other ways in which differential description might be implemented. One would be by using rationale-annotated cases of similar projects, such as those provided by the ARCHIE system [29]. Another way might be to use design patterns annotated with rationale.

Of course, differential description only works for domains where previous design work has been done and where someone has built collections of issue-based discussion, precedent cases, or design patterns. By definition, this approach is not useful for unprecedented problems. It should also be noted here that Rittel's theory of wicked problems, which led to the first DR method, included the notion that design problems are "essentially unique," and thus not easily solved by looking to precedents [54].

A number of researchers have explored ways of capturing DR without use of any schema, either because schemas are too labor intensive to use or because they interfere cognitively with capture. For example, Shipman and his collaborators from Xerox PARC built "spatial hypertext systems" [61] that enable informal input of information in a 2D space and then infer the structure of that information from its spatial arrangement, work inspired in part by gIBIS's graphical representation of IBIS structure. Reeves [52] also created a system that uses a schema-free approach to capture. With his system designers write their rationale as textual notes in the graphical representation of a physical artifact in a CAD system. The design history of the artifact then becomes the means by which rationale is structured. A different schema-free and completely nonintrusive approach is used by Myers et al. [47]. They add semantic information to a CAD system's symbol library and then infer the DR from the designer's use of the system. This approach, however, does not produce argumentation as such.

The idea of abandoning use of an explicit schema is controversial in the DR field. On one side of the debate, there are MacLean et al. [38] arguing for intrusive, schema-based approach to DR capture. At the opposite end are Shipman and Marshall [57, 58, 59] arguing for nonintrusive approaches that abandon use of schemas.

Another approach to facilitating DR capture tries to find when rationale is naturally elicited as part of design communication [60]. In these cases eliciting DR is not an extra task for designers and does not interfere with design. It is instead an already existing and accepted part of the design process. In fact, it is the means by which collaboration takes place in design. There are two approaches that can be taken in using design communication as the basis for DR capture. One is to structure that communication using a schema. Another approach is to record it in its natural, informal form and

then structure it retroactively, for example, by using natural language processing [45].

1.6 Rationale Management in Software Engineering

This section focuses on rationale-based approaches specific to software engineering (SE). First, we describe different types of SE projects where the use of rationale could have the most benefits, that is, in which addressing the limitations described in the previous section could yield a significant return on investment. In Sect. 1.6.2, we analyze for each SE activity how rationale can be captured and used. Using the SPICE process standard as a framework for discussing activities [http://www.sqi.gu.edu.au/spice/], we conclude that the concept of DR is too limited to encompass all the rationale-based processes of SE. We suggest that the concept of *SE rationale (SER)* is more general and more useful for discussing rationale management in SE. In Sect. 1.6.3, we present representative SER management approaches. Finally, we summarize the different SER approaches by activity, usage, schema, and original features.

1.6.1 Opportunities for Rationale in Software Engineering

Despite the challenges to the capture of rationale discussed in Sect. 1.5, there are also specific contexts in SE where the benefits of rationale capture could outweigh the costs. Below we list four contexts suitable for the four categories of uses introduced in Sect. 1.4.

- *Distributed projects*. A current trend of SE projects is the outsourcing of development, sometimes to organizations that are in different time zones. This leads to a breakdown in informal communication, where rationale is usually communicated peer-to-peer. Thus, approaches that use rationale to support collaboration could help here.
- *Product-line projects*. As products become instances of a product line, the life cycle of the product line becomes longer and the number of products that impact its design is high. Rationale can then be used to relate features of the product lines to specific product needs. It also can be used to externalize knowledge to guard against staff turnover. This could be alleviated by rationale uses focusing on reuse and change.
- *Safety critical systems*. Traceability of decisions is an important prerequisite for high-quality decisions, in particular when dealing

with change requests. Some organizations such as EUROCONTROL require this explicitly. Rationale can support this traceability. Clearly, rationale focusing on quality is most valuable here. Furthermore, in this context the high cost of failure changes the perception of the cost involved in rationale management.

– *COTS-based or mobile systems*. When systems are assembled from existing parts (either at deployment time of even at execution time in case of mobile systems) rationale can be useful to externalize knowledge between customer and supplier. Approaches focusing on knowledge transfer are most valuable here.

1.6.2 Supporting Software Engineering with Rationale

As described in Sect. 1.4, there are many different uses of rationale. Clearly, rationale can be provided on all decisions during SE. According to the sketch of the new version to be published in summer 2006, SPICE distinguishes the following process areas:

– Acquisition and supply (CUS1 and CUS2)
– Engineering (CUS3, ENG1, and ENG2)
– Operation (CUS 4)
– Support Processes (SUP1 to SUP8)
– Management (MAN1 to MAN4)
– Reuse (ORG6)
– Process improvement (ORG1, ORG2, and ORG5)
– Resource and infrastructure (ORG3 and ORG4)

In the following, we describe these process areas and analyze how rationale management could support ongoing or future activities.

When we defined DR in Sect. 1.2.1, we used a broad definition of the design process. But no reasonable definition of design is broad enough to encompass all the processes described in SPICE. Many of these processes involve decision making that is not part of design. If we consider carefully how the rationale associated with these decisions is generated and used, it becomes clear that the concept of *DR* is not broad enough to include all the rationale that needs to be managed in SE

We have defined the term *DR* in a way that corresponds to how it is typically defined in the literature. In this definition, DR includes two things: (1) rationale generated by designers, regardless of who makes use of that rationale and (2) rationale used by designers, regardless of who generates that rationale. Thus, if rationale generated by designers is used by software maintenance personnel, it is typically called DR. If rationale

generated by software maintenance personnel is used by designers, e.g., to design a future version of the software, then it is also typically called design rationale.

This definition of DR might seem to suggest that the only rationale in SE is design rationale. But we can see that this is not so by looking carefully at decisions taken in the nondesign processes of an SE project, for example, decision made in software maintenance. The people who make such decisions often do so, on the basis of explicitly stated rationale. Some of this rationale might be useful for design, as indicated earlier, and we could, according to our definitions, count this as DR. It is quite possible, however, that some of the rationale is only useful for maintenance, e.g., for keeping track of which maintenance decisions have been made and what the justification was for these decisions. In this case, the label *DR* is not appropriate. We would have to call this something like *maintenance rationale*. Once we acknowledge the legitimacy of such a term, however, it seems that some of what we have called *DR* might with equal justification be labeled *maintenance rationale*. By extension we can see that every process within SE has an equal claim to having rationale of its own.

Since design is only one of many SE processes, the term *DR* is not general enough to encompasses all the types of rationale that a rationale management systems needs to deal with in SE. In the following sections we will therefore use the term *software engineering rationale* (SER) to encompass all these different types of rationale. We use the term *rationale* when we do not specifically address the difference between SER and DR.

There is every reason to expect that the discussion on DR (SER for design) presented in the previous sections will be true for every other kind of SER. There may, however, turn out to be additional facets.

Acquisition and Supply

Acquisition encompasses the preparation (in terms of definition of criteria and provision of resources), the selection of a supplier, the monitoring of the supplier during engineering, and the acceptance of the product through the customer. Supply mirrors these activities on the side of the supplier. SER on the current customer system and on the supplied components supports the communication between customer and supplier.

During acquisition preparation, SER of the current customer system could help to understand the current software and its limitations (both by the customer and the supplier). In particular, the decision about whether to extend the current system or to buy a new one would be facilitated.

During supplier selection, SER of the supplied components could be used to justify how the components satisfy the acquisition requirements. The same holds true during supplier monitoring and the acceptance test.

Engineering

Engineering encompasses the entire development process including software requirements analysis, software design, construction, and integration, as well as software testing, system integration, and testing, as well as system and software maintenance.

In this process area SER essentially improves the communication between stakeholders and the quality of the products (Fig. 1.1). By communication, we mean in particular elicitation of knowledge, any kind of negotiation and structuring of meetings. With quality we mean consistency and correctness of decisions with respect to decision criteria, including automatic checks.

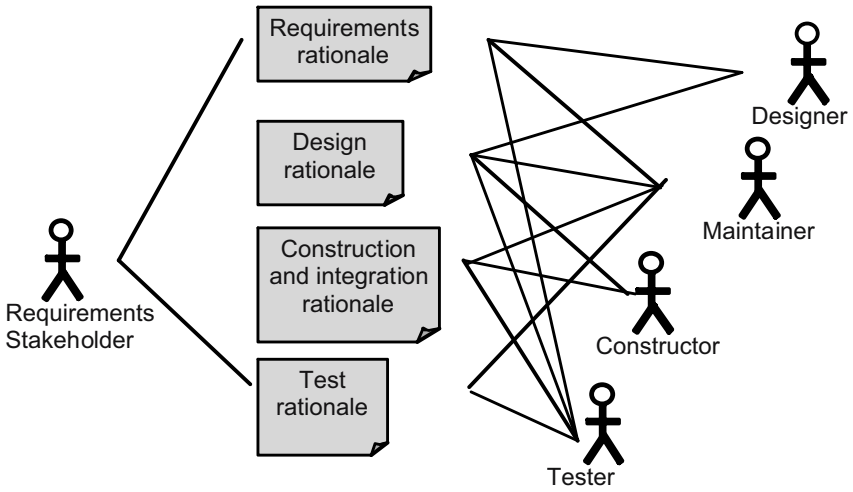

Fig. 1.1. Engineering rationale use

As for acquisition, preparation rationale about the current system supports the shaping of requirements on a new system. Rationale on the elicited requirements supports communication between the requirements stakeholders and to the designers, maintainers, and testers. SER of the design supports the (automatic) verification of the design against the requirements, and the communication between the designers and the constructors, maintainers, and testers. Similarly, SER of the construction supports the (automatic) verification of the construction against the design and the communication between constructors and the maintainers and testers. In

the same manner, integration can be verified against requirements and design, and needs to be communicated between customers and suppliers (in particular, testers). During testing, SER can be captured with respect to test coverage decisions. This can be used to verify that tests cover all requirements and to support communication between testers and to maintainers. Maintenance is one of the most popular rationale usage areas so far. As discussed in [12, 15] maintenance decisions concern sensitivity analysis with respect to possible changes, pretraceability to identify what prompted the change in the system element, posttraceability to identify what is influenced by the system element and impact analysis to identify the consequences of a change. All of these decisions profit from SER about the system being maintained. In addition, SER rationale on the maintenance actions helps to verify that the changed system meets its requirements and supports the communication between customers and suppliers on the changes.

Operation

During operation, both the customer and those supplying support need to understand how the system will behave. Both can be aided in gaining this understanding by SER captured during engineering.

Support

Support encompasses a number of specific processes within the engineering process area, namely, documentation, configuration management, quality assurance, verification, validation, joint review, audit, and problem resolution. As mentioned with respect to engineering, verification is supported by rationale. The same holds true for validation, which tries to show that the systems meets the user expectations. For the review of a product (e.g. requirements or design) the rationale concerning the product facilitates understanding by the reviewers. Similarly, for the audit of a process, the rationale about that process facilitates understanding by the assessors. As a special case, quality assurance ensures that the processes required by the customer were followed and the required artifacts were produced. SER could be used to justify why specific processes were not executed or why certain artifacts were not provided. As SER is particularly well suited for making alternatives explicit, it could facilitate configuration management by making configuration options explicit and enabling automatic configuration. Furthermore, SER helps to make argumentation from differing points of view explicit. This could be used to generate documentation for

people with fundamentally different perspectives on the SE process, e.g., different stakeholders.

Management

SER captured in the above-mentioned process areas can be used to support communication with management, as management needs to understand the forces that have lead to special project situations. For example, the number of unresolved issues or the priority of certain requirements can serve as indicators of project status. SER produced during project management could focus on risks. This would support both the communication about and the evaluation of those risks.

Reuse

Software reuse is another popular usage of rationale. The SER of any artifact produced or of any process step can indicate the situations in which the artifact or step is reusable and in what way it is reusable. This kind of SER is typically consolidated to enable quick and informed reuse decisions. One popular example where SER is crucial for a reusable artifact is a design pattern.

Process Improvement

Clearly, SER is very beneficial during process improvement. During process establishment, SER for specific process facets can increase the acceptance of the process. During process assessment, SER will support understanding by the assessors. Consolidated SER (e.g., as a part of patterns) can be used to suggest new process steps (see also reuse).

Resource and Infrastructure

Finally, the SER of any artifact or process step part of the current project helps newcomers to understand the current situation. This is particularly helpful for new employees to become quickly involved in the team.

1.6.3 Survey

In Sect. 1.6.2, we analyzed the potential use of SER for each SE activity. In this section, we present selected rationale approaches for SE, illustrating the current state-of-the-art for each activity. Our goal is not to provide a complete survey, but rather to select representative examples illustrating how the limitations described in Sect. 1.5 can be addressed. While most SE

activities could benefit from SER, current research has focused mostly on engineering, management, reuse, and process improvement.

Engineering

Eliciting Requirements – SCRAM
SCRAM [62] is a requirements elicitation approach combining several techniques, including scenario-based requirements elicitation and QOC. To elicit requirements, end users are presented a mockup of the system in the context of a usage scenario. Next, SER of key aspects of the mockup is shown to end users as a QOC model, emphasizing its advantages and weaknesses compared to other alternatives with respect to a set of criteria that have been identified so far. By making the SER explicit to the end users, requirements engineers not only can evaluate the current solution, but also elicit additional criteria and priorities among criteria. In general, the presentation of several options provided more discussion opportunities for end users and resulted in more kinds of information being elicited. Thus, collaboration and knowledge transfer is enhanced. SCRAM differs from Design Space Analysis in that recording SER is not a long-term goal in itself, but rather, a short-term means for eliciting additional knowledge from the client.

Elaborating Requirements – Inquiry Cycle
The Inquiry Cycle is a general process model for requirements elaboration [50]. It includes three activities, *expression*, *discussion*, and *commitment*, which are repeated in sequence. During the expression activity, stakeholders acquire domain-related knowledge, propose new requirements or scenarios. During the discussion activity, stakeholders comment and annotate the proposed requirements. During the commitment activity, stakeholders make decisions, generate change requests, or commit to find missing information. The cycle is repeated as often as necessary. Tool support for the Inquiry cycle included IBIS-like support for discussions, allowing stakeholders to track questions, answers, reasons, and requirements within the same tool. Like SCRAM, rationale is used for eliciting more information from stakeholders (as opposed to capturing long-term rationale). Unlike SCRAM, the Inquiry Cycle focuses on asynchronous and ad hoc use of SER, as opposed to post hoc use of a design space.

Refining Nonfunctional Requirements – NFR Framework
The NFR Framework [10] is a method for tracking the relevant nonfunctional requirements for each decision, evaluated alternative, and interaction among nonfunctional requirements. Nonfunctional requirements are

treated as goals to be met. To address the difficulty that nonfunctional requirements are usually high-level and subjective, goals are refined and clarified by decomposing them into subgoals. Goals and subgoals are represented as nodes in a goal graph. Decomposition relationships are represented as directed arcs. The NFR Framework provides two types of decompositions:

- *AND decomposition*. A goal can be decomposed into subgoals, all of which need to be met to help the parent goal.
- *OR decomposition*. A goal can be decomposed into alternative subgoals, any one of which needs to be satisfied to help the parent goal.

The top-level goals (specified by the client and the users) are hence refined by developers into lower-level and more concrete goals. Note that a single subgoal can be related to more than one parent goal. Moreover, the NFR Framework provides additional types of links to capture other relationships. For example, correlation links between two goals indicate how one goal in the graph can support or hinder the other goal. Since nonfunctional requirements are rarely qualities that are either met or not, links in a goal graph represent how much a goal contributes to or hinders another goal. A goal is satisficed (as opposed to satisfied) when the selected alternative meets the goal within acceptable limits. Otherwise, the goal is said to be denied. Root nodes represent high-level goals specified by the client. As these goals are refined into more concrete ones the refinement activity moves toward system features. Goals that represent system features are called operationalizing goals.

The NFR framework enables stakeholders to evaluate trading off different options against a set of conflicting criteria. By the end of the refinement process, the stakeholders can record the selected option as well as the explored alternatives and their reasons for not selecting them.

Tracing to Human Sources – Contribution Structures

Contribution structures [22] record the authors of requirements and their role in shaping the requirement, so that the originators of requirements can be identified, or, minimally, their intent better understood, when requirements are changed. The contribution structures framework distinguishes three capacities:

- The *Principal* motivates the requirement and is responsible for its effects and consequences.
- The *Author* develops the requirements' structure and content and is responsible for its form and semantics.

– The *Documentor* records or transcribes the requirements' content and is responsible for its appearance.

Recording the role of a contributor with respect to a requirement provides a simple way to document the commitment and responsibility of the contributor. Although contribution structures do not capture an explicit intent in the way IBIS or QOC does, traceability to human sources enable change requirements prioritization and change requests to be directed to the right contributor, based on the nature of the change and the requirements being changed.

Post-Traceability – REMAP
REMAP is a conceptual model extending IBIS to include requirements and design elements to process knowledge during requirements engineering [51]. A prototype of REMAP was built to demonstrate how requirements, design elements, design decisions, constraints, and argumentation are captured in a graph, representing the process by which requirements and design were generated and negotiated. Using a truth maintenance system, the REMAP prototype propagates constraints and the validity (or invalidity) of assumptions through the graph, illustrating the benefit of traceability from requirements through SER and design elements. REMAP is in essence similar to DRL.

Requirements Checking – C-ReCS
C-ReCS is a tool for supporting collaborative requirements and recording decisions [27, 28]. It enables users to specify requirements and their SER in a formal language, a semantic net composed of predefined entities. The tool then provides users a suite of tools for detecting, diagnosing, and proposing resolutions for exceptions, such as consistency, completeness, and correctness problems. Once an exception is detected, the diagnosis attempts to explain to the user the underlying cause of the exception, using a predefined decision tree.

For example, C-ReCS detects inconsistencies based on the propagation of constraints in the requirements graph. A diagnosis would then present the propagation trace and the two constraints that are in conflict. This in turn serves as a basis for suggesting that the user relaxes one or the other constraint. When changing the requirements to remove the inconsistency, the user can link to the diagnostic as SER for the change.

Design Checking – SEURAT
SEURAT is a tool for recording and using SER of the system under construction at the level of source code [5]. SEURAT is integrated into the development environment, making it easier to switch back and forth

between development and documentation tasks. It is based on an extension of DRL, allowing the representation of detailed arguments and dependencies. It also provides a rich ontology of arguments, making it easy for the developer to reuse arguments. The ontology, combined with rules for syntactic and semantic checking, enables SEURAT to automatically identify inconsistencies or omissions in the rationale.

The rich and extensible argument ontology aims at lowering the effort for developers to capture SER, while increasing its accuracy. Making it accessible in a development environment, and providing services that are similar to standard style and consistency checking on source code, SEURAT also aims at increasing the short-term incentives for developers to use the SER they provided.

Long-term Collaboration – Sysiphus
Sysiphus provides a simple and integrated solution to manipulate system models and SER, embedding only minimal process specific knowledge [14,16]. This allows different development processes and the use of SER for a broad range of activities. Sysiphus includes a tool suite centered on a repository, which stores all models, SER, and user information. The repository controls access and concurrency enable multiple users to work at the same time on the same models. SER elements are first class objects (as opposed to buried notes or comments) and are accessed the same way as system model elements. The tool puts equal focus on the system and the SER. The end user can browse back-and-forth between SER and system models. Changes made by the end user are propagated synchronously to other end users working on the same model, enabling users to collaborate synchronously. When overlaps are discovered, the end user is prompted by the system to merge conflicting changes.

Sysiphus adopts a similar approach to SEURAT for lowering the threshold for capturing SER and increasing short-term developer incentive. However, Sysiphus focuses on the modeling and collaboration environment while SEURAT focuses on the development environment.

Management

WinWin
WinWin [3] is an approach where SER is used in support of risk management. WinWin resulted from the observation that satisfying all key stakeholders is a necessary condition for project success. Often, the issue of dealing with conflicting success criteria is not only to reconcile conflicting views, but also to identify the key stakeholders of the system and to clarify their success criteria. Once these criteria are known to all, it is much easier

to identify conflicts and to resolve them by negotiating compromise alternatives.

The WinWin negotiation model, similar to the QOC model described in Sect. 1.3, includes four elements. *Win conditions* are criteria, originated by stakeholders that, if not met, result in the failure of the project. *Issues* represent areas of disagreements typically a conflict between Win conditions that need to be further clarified or negotiated. *Options* represent alternatives for resolving issues, and *Agreements* represent decisions for closing an issue. Finally, Win conditions and agreements are classified into taxonomy categories. The taxonomy is specific to the system under construction and is used to relate large numbers of win conditions and agreements to broad requirements categories.

WinWin is tightly integrated into Boehm's spiral model. For the each iteration, critical stakeholders are identified and the win conditions relevant to the current iteration are elicited and reconciled. Win conditions are prioritized and scheduled to iterations based on risk. For example, a strong area of disagreement can result in a small set of win conditions being addressed in an early iteration, to ensure that an area of agreement can be found and to build trust among stakeholders.

Reuse

Augmenting Design Patterns with Rationale – DRIMER

DRIMER is a software development process and tool for applying design patterns [48]. Developers can search a design pattern catalog based on their intents, and examine specific examples of use of the design pattern. SER for each example is also provided following the DRIM schema, making it easier for a novice developer to understand unfamiliar patterns and for the experts to validate their usefulness. DRIM is similar to DRL, provides elements for representing intents, proposals, recommendations, justifications, and context of decisions. By integrating the process of finding reusable solutions with the process of recording experiences, DRIMER aims to create short-term incentives for developers to provide SER information while lowering the effort involved with capturing it.

Process Improvement

CoMoKit

CoMoKit is a process modeling and enactment tool that automatically records dependencies among products [11]. A process model specifies how products are generated and used by tasks. Tasks can be refined into subtasks, all of which need to be completed for the parent task to be

completed. The process model can include several methods for accomplishing the task, each possibly resulting in different products.

The approach assumes that there is a causal relationship between the input products of a task and its outputs. When the process model is enacted (i.e., when the user executes tasks, selects methods to create products), the tool records causal dependencies between products and decisions. Moreover, the user can add additional justifications for or against decisions.

When decisions or products are invalidated, CoMoKit automatically retracts other decisions and products that were derived from the newly invalidated element.

CoMoKit is similar to REMAP, in that it captures dependencies between products and decisions, and uses a truth maintenance system to propagate validity. Unlike REMAP, CoMoKit captures some dependencies automatically and provides a unified representation for both user-specified and generated rationale.

1.6.4 Summary

A summary of the SER approaches surveyed in this section is given in Table 1.1. Much progress has been made on the development of such

Table 1.1. Summary of SER approaches

Approach	Schema	SE activity	rationale use
SCRAM (1995)	QOC	requirements elicitation	collaboration
Inquiry Cycle (1994)	IBIS	requirements elaboration	collaboration
NFR Framework (1999)	Goal graph	nonfunctional requirements refinement	improve quality
Contribution structures (1994)		requirements change	collaboration
REMAP (1992)	IBIS++	requirements management	improve quality
C-ReCS (1997)	DRCS	requirements elaboration	improve quality
SEURAT (2004)	DRL++	development	improve quality
Sysiphus (2001)	IBIS/QOC	any	collaboration
WinWin (1994)	IBIS	risk management	collaboration
DRIMER (1996)	DRIM		reuse
CoMoKit (1996)		process improvement	improving quality

approaches and tools since the early 1980s. A number of important prototypes have been developed, but few rationale management systems have made it into practical use in industry. Recent research tends to combine these systems with other forms of design support systems [2, 26].

1.7 Tool Support for Rationale Management

This section describes an ideal tool support for rationale management in SE (rationale management system, RMS for short). In Sect. 1.7.1 ,we describe the life cycle of SER knowledge, which is used to deduce the functional requirements of an RMS. In Sect. 1.7.2, we discuss further requirements to overcome the challenges identified in the earlier sections. In Sect. 1.7.3, we describe a generic RMS in terms of an architectural framework populated by a set of components.

1.7.1 Rationale Life Cycle

In Sect. 1.4, we described the uses of rationale, that is, the ways in which rationale adds value to a development project. When developing tool support for rationale management, however, we need to consider the entire lifecycle of rationale knowledge, from planning to preservation. In view of general knowledge management, we can identify the following rationale management tasks [13]:

- Rationale goal definition
- Rationale measurement
- Rationale identification
- Rationale acquisition
- Rationale development
- Rationale distribution
- Rationale use
- Rationale preservation

Rationale goal definition, measurement, and rationale identification are critical for identifying the kind of rationale needed, but they are strategic planning activities and, thus, are typically not supported by an RMS. However, the outcome of these activities is a critical prerequisite for deploying an effective RMS. We discuss this further in Sect. 1.7.2.

All the other tasks can be directly supported by an RMS. In the following, we list the required features:

- Rationale acquisition is most often called *rationale capture*. Here the major question is how rationale is captured, for example, through reconstruction, apprentice shadowing of designers, or automatic generation. Other possibilities include capture during communication and reasoning.
- Rationale development *structures and packages rationale*. The major question is how to represent rationale. Lee [36] identifies three layers as representative of a generic structure of an RMS:
  - A decision process layer which stores the rationale, e.g., into five sublayers: issue, argument, alternatives, evaluation, and criteria
  - A design artifact layer which links the rationale to the development process artifacts, e.g., a product-process model
  - A design intent layer: meta-information underlying design decisions, such as intents, strategies, goals, and requirements

Further questions are whether representations are informal, semi-formal, or formal and is visual modeling used [20, 21].

- Rationale distribution makes the *rationale available for concurrent users*. An important issue here is ease of retrieval e.g., through a user-adaptable feature to browse, view, and filter the rationale. This should also enable the answering of questions and the review of similar design cases. Another important issue is collaboration, as rationale is often captured during collaboration.
- To support rationale use, the RMS must be *closely integrated into the tool support for the SE tasks*. Furthermore, it should support reasoning about the available rationale and the development artifact, for example, evaluation of given artifacts based on their rationale or suggestions for enhancements and modifications of artifacts based on available rationale.
- To allow *long-term usage of the rationale*, the RMS should support *rationale preservation*, for example, by filtering out redundant rationale or by giving priority to rationale that has been critical during development.

The above features must be adapted to the context in which the RMS is used. For example, development could be more process-oriented or more feature-oriented at different development stages. In fields with a relatively high degree of understanding of problems, solution technologies and standardization of artifacts, the feature-oriented approach can be used to give logical representation of artifacts, to follow the rules of the process.

In development where the problems or solution technology are poorly understood and where there is little standardization of artifacts a process-oriented approach can provide historical representation of artifacts [9].

Ideally, the RMS should support these features throughout all SE activities. So far, however, RMSs have been most successful when adapted to specific activities and specific goals. These goals depend mostly on the rationale usages identified in Sect. 1.4.

1.7.2 Dealing With Rationale Challenges

In addition to supporting rationale tasks, an RMS must deal with the limitations discussed in Sect. 1.5. In particular, strategic decisions made during the rationale goal definition and rationale identification tasks can significantly impact the selection and tailoring of the framework components.

- *Assessing cost vs. benefits*. The project or organization must a priori identify areas in which the use of rationale can yield a return on investment. For a project developing a safety critical system, rationale may facilitate the safety analysis of the design. For a COTS-based project, selecting a COTS with its available rationale may reduce the effort for integrating it into the system.
- *Addressing the capture problem*. In addition to identifying what kinds of rationale should be captured, a means and incentive for capturing it must also be identified. This can range from schema-free capture and automated structuring using natural language processing or inference to demonstrating compliance with review certification criteria. Developers capturing rationale should have a clear short-term use or benefit for capturing it.
- *Dealing with scale and complexity.* The scale and complexity of captured rationale depends on the selected granularity, the scale and complexity of the system and application domain, and on the rate of change of decisions. Accordingly, the RMS needs to account for these issues, by providing the necessary traceability links, search, versioning, filtering, and customization features. Automating syntactical and semantic checks, such as in SEURAT, enforces a higher level of consistency in the captured rationale, especially when many end users are involved.

1.7.3 An Architectural Framework for Rationale Management

Tool support for rationale has been often viewed as a stand-alone system. A monolithic tool supports the capture, representation, and use of rationale, either as a general-purpose tool such as gIBIS or a tool specialized to an activity, such as CoMoKit for process enactment. Instead, we view a rationale system as supporting designers in handling designs within a

framework. An RMS is mostly transparent, appearing as an extension of the design environment, adaptable to the specific project situation.

In system terms, we propose that tool support for rationale should be viewed as a framework of components, each supporting a different activity and able to produce outcomes compatible with other components. In this section, we describe such a framework, its components, its interfaces to the design environment, and the constraints it must satisfy (Fig. 1.2).

Capture Components

An RMS supports rationale acquisition with a number of capture components for recording rationale from developers, extracting it from artifacts, or inferring it from developer actions. Such components might support:

- *Rationale capture by supporting collaboration*. Systems such as gIBIS, WinWin, or Sysiphus support project participants for communication and collaboration by providing a structured set of actions and entities for exchanging their opinions and criteria. In effect, the tool structures the collaboration to elicit the rationale to be captured and to reduce the overhead for structuring it. To increase collaboration through the component, many such SER components also provide a complete range of groupware features, such as group awareness, synchronous and asynchronous modes of communication, and support for multimedia.
- *Rationale extraction from artifacts.* An alternative approach is to extract rationale from communication or design artifacts after the fact. Natural language processing approaches identify key issues and arguments from natural language text, removing the burden from the participants to follow predefined schemas.
- *Rationale capture in design reasoning*. Systems such as SEURAT provide design support, either on their own or integrated into a larger development environment. This enables the capture of traceability links and inference of knowledge from the actions of the developer.
- *Rationale as justification*. Developers currently document rationale for decisions that are not obvious or that could impact other decisions. Systems like CoMoKit recognize the need for explicit capture of justifications and relate them with rationale captured or inferred by other components.

We expect that the most development projects will require a combination of the above components, depending on project-specific opportunities and constraints for capturing rationale.

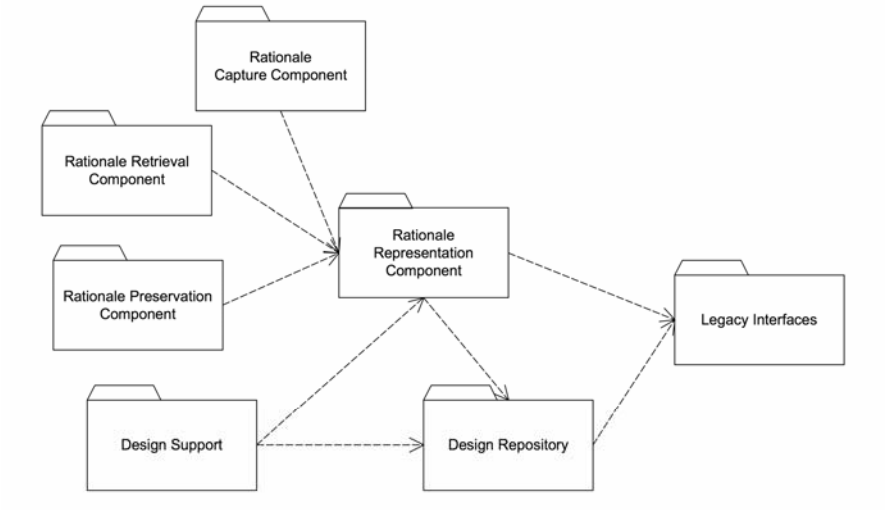

Fig. 1.2. RMS component overview

Representation Component

Even if the input of rationale is schema free and its formalization automated (see Sect. 1.5), an RMS supports rationale development with a representation component that provides a schema for storing and relating the rationale to other artifacts. A minimum amount of structuring is necessary for making it easier for developers to maintain, search, or relate to the design context. Most RMSs provide their own proprietary representation component, based on the specific SE activity that they support. A general RMS spans many activities and, as such, requires an open and extensible representation so that rationale can be captured at different levels of detail or be categorized according to different ontologies, based on the project context and activities supported. Such a representation could be used to enable different activities in the same environment to use either IBIS or QOC, organize issues hierarchically as in PHI, and capture intents as in DRL. A critical feature of the representation component is its ability to relate captured rationale to design artifacts, in particular, specific versions or configurations of the software, documents, or models.

Retrieval Components

An RMS supports access to rationale with retrieval components whose task is to derive information from rationale to facilitate their current task. Retrieval components range from simple generic components for navigating

the rationale to specialized components that check for design rule violations or that evaluate designs:

- General-purpose components
  - Retrieve by query
  - Navigate rationale
  - Visualize rationale
- Specialized components
  - Formulate design documents
  - Answer designer's questions
  - Identify similar design cases
  - Design reasoning
  - Evaluate design

Preservation Components

An RMS supports rationale preservation with components for restructuring and reformulating rationale for long-term use. For example, rationale captured from communication is often incomplete. Terminology evolves and specializes over the course of the project, making initial requirements rationale more difficult to understand. There is a need for explicit preservation components. There has been little research in this dimension of RMSs so far, because the attention has been focused so heavily on rationale capture.

Interfaces to Legacy Components

The primary focus of a designer is on the plan leading to the artifact. Developers produce system designs that lead to the construction of software. A project manager produces task plans that lead to the consumption of resources and the production of economic value for the project. Rationale is a support function and is not the main focus of the designer. Consequently, there has been a trend towards tight integration of design rationale representations with other design representations [5, 14, 27] with the RMS being treated as an extension of the design system. An RMS must be able to interface with many external artifacts and tools (Fig. 1.3):

- *Product history*. Rationale evolves with the system under construction. As the system changes, developers need to justify changes and update rationale already captured. Consequently, they need to link to the design repository, that means different versions and configurations of the system and its design when formulating justifications.

- *Knowledge base*. Organizations accumulate knowledge that lives across individual projects, in terms of guidelines, lessons learned, and standards. Such knowledge finds its source in actual cases and also serves as the basis for decisions in subsequent projects. An RMS should also provide the ability to link to and from this knowledge.
- *Patterns base*. Developers refine pattern solutions for recurring design problems. As such pattern solutions become more general and refined, it becomes necessary to document its possible usage and trade-offs encountered during their uses. By attaching rationale to pattern solutions, developers can more easily identify which pattern to apply and how to refine it. Similarly, linking design decisions with a patterns base avoids repeating this rationale in the design.
- *Process models and enactment*. Recording rationale (justification behind process-level decisions, as for example in CoMoKit [11]) similarly enables organizations to reuse and evolve processes. While we do not expect process and product rationale to overlap significantly, using a uniform environment for capturing both would reduce training overhead and increase familiarity among project participants.

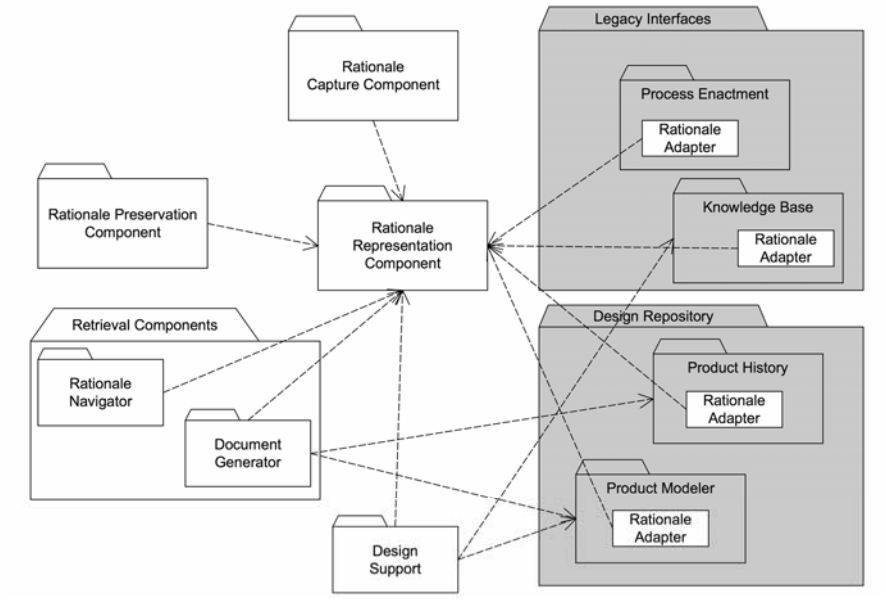

Fig. 1.3. An example of RMS architecture with legacy interfaces

1.8 Conclusion

In this chapter, we reviewed the state-of-the-art in rationale management in SE. We first provided a historical perspective by examining DR research in general. We then identified challenges and limitations faced by rationale approaches in SE. We also explained that to describe the rationale for all the processes of SE we need a more general term than *DR*; and for this purpose we adopted the term *SER*. We then discussed selected rationale approaches applied to SE, illustrating how specific challenges could be overcome. Finally, we presented an architectural framework for rationale management tool support.

Over the past decades, the research community has achieved some consensus on selected rationale research issues. For example, it is now widely accepted that having developers formalize the rationale for their decisions as they work is disruptive and that collaborative or post hoc approaches have better chances of capturing rationale. While general-purpose methods have not been widely adopted, specialized approaches addressing narrow problems have emerged, such as providing rationale with design patterns to facilitate their reuse, both in terms of design and DR.

As solutions are found for front-end issues, we anticipate that the research focus will include rationale preservation issues. For example:

Activity cross-pollination. Approaches presented in Sect. 1.6 often focus on a single use or activity of rationale. As rationale is used across several activities, the cost of capturing and training developers will be lower, relative to benefits. It is unclear, however, how to manage such overlaps.

Development environment integration. Parts of the research community have come to the consensus that rationale support should be tightly integrated into the development environment. First, rationale supports design and could be captured as a side-effect of the design methodology. Second, as system models and decisions are revisited, their accompanying rationale needs to be re-examined. This entails strong traceability between rationale and system models. There is a consensus that there should be a tight integration, but it is unclear, how to achieve it beyond specialized cases.

Rationale maintenance. An often-advertised benefit of rationale is to support changes, across time, staff turnover, and organizational boundaries. This means that rationale knowledge is also long-term knowledge that needs to be updated and consolidated as systems and designs evolve, and that contains obsolete knowledge that should be retired. Surprisingly, there is little research on rationale maintenance. As capture and structuring

methods become more successful, rationale maintenance in particular, and rationale preservation in general, will need to be explicitly addressed.

Rationale management research has made inroads in a broad variety of disciplines, both within and outside the field of SE. As the confronted issues become more systemic, the interdisciplinary character of rationale research will be a critical asset in finding solutions that work beyond specialized situations.

Acknowledgments. We are grateful to David Brown, Janet Burge, Manny Lehman, Philippe Palanque, and Debbie Richards for their constructive and detailed feedback. All remaining errors are our own.

References

[1] Alexander C, Ishikawa S, Silverstein M, King I, Angel S, Jacobson M (1977) A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Oxford

[2] Banares-Alcantara R, King JMP (1997) Design support systems for process engineering – design rationale as requirement for effective support. Comput. Chem. Eng. 21(3): 202–212

[3] Boehm B, Egyed A, Kwan J, Port D, Shah A, Madachy R (1998) Using the WinWin spiral model: A case study. IEEE Computer 31(7): 33–44

[4] Burge J (1998) Design rationale. Technical Report, Worcester Polytechnic Institute, Computer Science Dept., In: http://www.cs.wpi.edu/Research/aidg/DRRpt98.html (accessed: 02/17/2005).

[5] Burge J, Brown DC (2004) An integrated approach for software design checking using rationale. In: Gero, J (ed) Design Computing and Cognition '04. Kluwer Academic Publishers, Netherlands, pp. 557–576

[6] Carroll JM, Rosson MB (1996) Deliberated evolution: Stalking the view matcher in design space. In: Moran T P, Carroll J M (eds.) Design Rationale: Concepts, Techniques, and Use. Lawrence Erlbaum Associates, Mahwah, NJ, pp. 107–145

[7] Churchman CW (1967) Wicked problems. Guest editorial, Manage. Sci., 14 (4): 141–142

[8] Conklin J, Begeman M (1988) gIBIS: A hypertext tool for exploratory policy discussion. ACM Trans. Off. Inform. Syst. 4: 303–331

[9] Conklin J, Burgess-Yakemovic K C (1991) A process-oriented approach to design rationale. Hum.–Comp. Interact. 6: 357–391

[10] Chung L, Nixon BA, (1996) Dealing with change: An approach using non-functional requirements. Requir. Eng. J. 4: 238-–260

[11] Dellen B, Kohler K, Maurer F (1996) Integrating software process models and design rationales. In: Proceedings of 11th Knowledge-Based Software Engineering Conference (KBSE '96) September 25–28, Syracuse, NY, pp. 84–93

[12] Dutoit AH, Paech B (2000) Supporting evolution: Rationale in use case driven software development. In: Proceedings of the International Workshop on Requirements Engineering: Foundations of Software Quality (REFSQ'2000), Stockholm, June, pp. 99–112
[13] Dutoit AH, Paech B (2001) Rationale management in software engineering. In: Chang SK (ed.) Handbook of Software Engineering and Knowledge Engineering. Vol. 1. World Scientific, Singapore
[14] Dutoit AH, Paech B (2002) Rationale-based use case specification. Requir. Eng. J. 1: 3–9
[15] Dutoit AH, Paech B (2003) Eliciting and maintaining knowledge for requirements evolution. In: Aurum A, Jeffery R, Wohlin C, Handzic M (eds.) Managing Software Engineering Knowledge. Springer, Berlin, pp. 135–156
[16] Wolf T, Dutoit AH (2004) A rationale-based analysis tool. 13th International Conference on Intelligent and Adaptive Systems and Software Engineering, July 1–3, Nice, France
[17] Fischer G, Lemke A, McCall R, Morch A (1996) Making argumentation serve design. In: Moran TP, Carroll JM (eds.) Design rationale: Concepts, techniques, and use. Lawrence Erlbaum, Mahwah, NJ, pp. 267–294
[18] Fischer G, McCall R, Morch A (1989) Design environments for constructive and argumentative design. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Wings for the Mind, March, ACM New York, pp. 269–275
[19] Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns. Addison-Wesley, Reading, MA
[20] Ganeshan R, Garrett J Jr, Finger S (1994) A framework for representing design. Int. J. Design Stud. 1 : 59–84
[21] de la Garza J, Alcantara P (1997) Using parameter dependency network to represent design rationale. J. Comput. Civil Eng., 2(2): 102–112
[22] Gotel O, Finkelstein A (1995) Contribution structures. In: Proceedings International Symposium on Requirements Engineering, IEEE, York, pp. 100-–107
[23] Gruber TR, Russell DM (1996) Generative design rationale: Beyond the record and play paradigm. In: Moran TP, Carroll JM (eds.) Design Rationale: Concepts, Techniques, and Use. Lawrence Erlbaum, Mahwah, NJ, pp. 323–349
[24] Grudin J (1988) Why CSCW applications fail: problems in the design and evaluation of organization of organizational interfaces. In: Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work, ACM, New York, pp. 85–93
[25] Grudin, J (1996) Evaluating opportunities for design capture. In: Moran TP, Carroll JM (eds.) Design Rationale: Concepts, Techniques, and Use., Lawrence Erlbaum, Mahwah, NJ, pp. 453–470
[26] King JMP, Banares-Alcantara R (1997) Extending the scope and use of design rationale. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 11(2): 155–167
[27] Klein M (1993) DRCS: An integrated system for capture of designs and their rationale. In Gero J (ed.) Artificial Intelligence in Design '92. Kluwer Academic Publishers, Boston, pp. 393–412

[28] Klein M (1997) An exception handling approach to enhancing consistency, completeness and correctness in collaborative requirements capture. Concurrent Engineering Research and Applications, 5(1): 37–46
[29] Kolodner J (1993) Case-based reasoning. Morgan Kaufmann, San Mateo, CA
[30] Kunz W, Rittel H (1970) Issues as elements of information systems. Working Paper 131, Center for Urban and Regional Development, University of California, Berkeley.
[31] Lai K, Malone T, Yu K (1989) Object lens: A 'Spreadsheet' for cooperative work. ACM Transaction on Office Information Systems, 6(4): 332–353
[32] Lee J (1990) SIBYL: a tool for managing group design rationale. In: Proceedings of the ACM Conference on Computer-supported Cooperative Work, ACM, New York, pp. 79–92
[33] Lee J (1990) SIBYL: A qualitative decision management system. Artificial intelligence at MIT expanding frontiers. MIT Press, Cambridge, MA
[34] Lee J (1991) Extending the Potts and Bruns model for recording design rationale. In: Proceedings of the 13th International Conference on Software Engineering (ICSE'13), IEEE Computer Society Press, Los Alamitos, CA, pp. 114–125
[35] Lee J, Lai K (1996) What is design rationale? In: Moran TP, Carroll JM (eds.) Design Rationale: Concepts, Techniques, and Use. Lawrence Erlbaum Associates, Mahwah, NJ, pp. 21–52
[36] Lee J (1997) Design rationale systems: Understanding the issues. AI in Design, IEEE Expert, May/June: 78–85
[37] Lewis C, Rieman J, Bells B (1996) Problem-centered design for expressiveness. In: Moran TP, Carroll JM (eds.) Design Rationale, Concepts, Techniques and Use, Lawrence Erlbaum Associates, Mahwah, NJ, pp. 147–184
[38] MacLean A, Young RM, Bellotti VME, Moran T (1996) Questions, Options and Criteria. In: Moran TP, Carroll JM (eds.) Design Rationale, Concepts, Techniques and Use, Lawrence Erlbaum Associates, Mahwah, NJ, pp. 53–106
[39] McCall R (1989) MIKROPLIS: A hypertext system for design. Des. Stud. 10(4): 228–239
[40] McCall R (1991) PHI: a conceptual foundation for design hypermedia. Des. Stud. 1: 30–41
[41] McCall R (1979) On the structure and use of issue systems in design., Doctoral Dissertation 1978, University of California, Berkeley, University Microfilms
[42] McCall R, Bennett P, d'Oronzio P, Ostwald J, Shipman F, Wallace N (1990) PHIDIAS: A PHI-based design environment integrating CAD graphics into dynamic hypertext. In: Rizk A, Streitz N, André J (eds.), Proceedings of the European Conference on hypertext (ECHT'90), INRIA, France, Cambridge University Press, New York, NY, pp. 152–165
[43] McCall R, Bennett P, D'Oronzio P, Oswald J, Shipman FM III, Wallace N (1992) PHIDIAS: Integrating CAD graphics into dynamic hypertext. In: Streitz N, Rizk A, André J (eds.), Hypertext: Concepts, Systems and Applications, Cambridge University Press, New York, NY, pp. 152–165

[44] McCall R, Mistrík I, Schuler W (1981) An integrated information and Communication system for problem solving. In: Proceedings of the Seventh International CODATA Conference, Kyoto, Japan, pp. 107–115
[45] McCall R, Mistrík I (2005) Capture of software requirements and rationale through collaborative software development. In: Maté JL, Silva A (eds.) Requirements Engineering for Sociotechnical Systems. Information Science Publishing, pp. 303–317
[46] Moran TP, Carroll JM (1996) Design Rationale: Concepts, Techniques and Use, Lawrence Erlbaum Associates, Mahwah, NJ
[47] Myers KL, Zumel NB, Garcia PE (1999) Automated rapture of rationale for the detailed design process. In: Proceedings of the Eleventh National Conference on Innovative Applications of Artificial Intelligence (IAAI-99), AAAI Press, Menlo Park, CA, pp. 876–883
[48] Pena-Mora F, Vadhavkar S (1996) Augmenting design patterns with design rationale. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 11: 93–108
[49] Potts C, Bruns G (1988) Recording the reasons for design decisions. In: Proceedings of the 10th International Conference on Software Engineering (ICSE'10). Los Alamitos, CA, pp. 418–427
[50] Potts C, Takahashi K, Anton A (1994) Inquiry-based requirements analysis. IEEE Software, March,: 21–32
[51] Ramesh B, Dhar V (1992) Supporting systems development by capturing deliberations during requirements engineering. IEEE Trans. Softw. Eng. 18 (6): 498–510
[52] Reeves B, Shipman FM III (1992) Supporting communication between designers with artifact-centered evolving information spaces In: Proceedings of the 1992 ACM Conference on Computer-supported Cooperative work, November 1–4, Toronto, Ont., Canada, pp. 394–401
[53] Rittel H (1985) personal communication
[54] Rittel H, Weber M (1973) Dilemmas in a general theory of planning. Policy. Sci. 4: 155–169
[55] Rittel HWJ (1972) On the planning crisis: Systems analysis of the first and second generations. Bedriftsokonomen, Norway, 8:390–396
[56] Schön D (1983) The reflective practitioner. How professionals think in action. Temple Smith, London
[57] Shipman FM III, Marshall CC (1999) Formality considered harmful: Experiences, emerging themes, and directions on the use of formal representations in interactive systems. Comput. Support. Cooperat. Work 8(4): 333–352
[58] Shipman FM III, McCall R (1994) Supporting knowledge-base evolution with incremental formalization. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Boston, Massachusetts, US, pp. 285–291
[59] Shipman FM III (1993) Supporting knowledge-base evolution with incremental formalization. PhD dissertation, Technical Report CU-CS-658-93, Department of Computer Science, University of Colorado, Boulder

[60] Shipman FM III, McCall R (1997) Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication. Artif. Intell. Eng. Des., Anal., Manuf., 11(2): 141–154
[61] Shipman FM III, Marshall C (1999) Spatial hypertext: An alternative to navigational and semantic links. ACM Computing Surveys, ACM, New York, 31(4es): 14
[62] Sutcliffe A, Ryan M (1998) Experience with SCRAM, a scenario requirements analysis method. In: Proceedings of the 3rd International Conference on Requirements Engineering, Colorado Springs, CO, pp. 164–173
[63] Toulmin S (1958) The Uses of Argument. Cambridge University Press, UK