4 Learning from Rationale Research in Other Domains

While the issues of rationale usage in software engineering (SE) often differ crucially from those of rationale usage in other domains, there is still the possibility of learning a great deal from research on other domains. This is suggested by the fact that rationale research in SE originally derived from Rittel's much earlier rationale research in architecture (building design), urban planning, and policy making. In addition to this work, which is still not widely known in SE circles, there is research on rationale that has been going on in various engineering disciplines for as long as 20 years. All of this work provides potentially valuable lessons for SE researchers and developers. This chapter will look at some examples of this work that could have important implications for rationale research in SE.

4.1 Introduction

4.1.1 Research on Rationale in other Domains

Research on design rationale began with Rittel's Issue-Based Information System (IBIS) (Kunz and Rittel 1970) and its applications to urban planning, architecture (building design), and governmental policy making in the 1970s and 1980s. By the late 1980s software engineers at the Microelectronics and Computer Technology Corporation (MCC) were looking at adapting Rittel's method to their own field and developing appropriate computer support (Conklin and Begeman 1988; Potts and Bruns 1988). Since then many other researchers involved with software engineering (SE), human–computer interaction (HCI), and other software-related related fields have created various rationale approaches, including QOC (MacLean, Young and Moran 1989), DRL (Lee 1991), RATSpeak (Burge and Brown 2004), and many others. Most of these approaches continue the basic tradition started by

Rittel, while suggesting various modifications meant to go beyond Rittel's IBIS and better fit rationale to the SE domain.

Chapter 2 of this book emphasizes that there are some crucial differences between the problems of rationale usage in the SE domain and rationale usage in the domains of physical artifact creation. At the same time, there continues to be a considerable overlap in the issues facing rationale researchers in these two types of domains. This suggests that researchers in these domain types might still have much to learn from each other. This chapter explores this topic by presenting some examples of rationale research in design and engineering.

4.1.2 Objectives of This Chapter

Rather than attempting a comprehensive survey of rationale research in other domains, this chapter will concentrate on examples of such research that raise important issues for research on rationale support in SE. For these examples, the issues raised mostly have to do with the way in which they use computers to support rationale; therefore, this chapter will go into more detail on the rationale management software systems than is generally the case in the remainder of the book.

The approaches and systems described in the chapter all deal with the rationale for design. For the examples discussed, this chapter will first identify crucial functionality that they bring to the support of rationale, functionality not currently found in rationale management systems for SE. Connections to existing research on software engineering rationale (SER) will then be identified. The potential advantages of adopting this functionality in SER support systems will then be discussed; and potential challenges to implementing this functionality in SE will be described.

4.2 Domain-Oriented Design Environments Using PHI

4.2.1 PHIDIAS and JANUS

The PHIDIAS (PHI-based Design Intelligence Augmentation System) project (McCall et al. 1990) began in 1985 with the goal of adding a CAD subsystem to the MIKROPLIS hypertext software. MIKROPLIS (McCall et al. 1981; McCall 1991) was a hypertext authoring system devised in the

early 1980s to support the PHI variant (McCall 1991) of IBIS (Kunz and Rittel 1970). In this project, fundamental issues arose about how the integration of CAD graphics and PHI rationale should work from the standpoint of human–computer interaction (HCI). These issues were ultimately settled not by working directly on PHIDIAS but by working on the JANUS system.

JANUS combined the functionality of the CRACK system (Fischer and Morch 1988) for kitchen design with hypertext functionality needed for PHI-based design rationale. CRACK enabled designers to create kitchen layouts using a *domain-oriented construction kit*. A construction kit is a collection of graphical building blocks that can be dragged and dropped into place in a CAD system. A construction kit is *domain oriented* if its building blocks represent high-level domain concepts, such as walls, windows, stoves, sinks, etc. rather than low-level computer graphics concepts such as points, lines, and shapes. Domain-oriented construction kits were used because they enabled designers to rapidly and intuitively build designs. Such a construction kit is, in essence, simply a conventional CAD *symbol library* to which semantics had been added so that each type of building block indicates what type of real-world object it denotes—e.g., window, door, stove or sink.

In CRACK the semantic information of the building block is used by a critiquing system that "looks over the shoulders" of designers as they work and points out violations of rules of thumb for kitchen design. An example of such a critique might be, "do not put the stove in front of a window." The rationale for this critique is that placing a stove in front of a window creates several potential problems: (1) a person might reach over the stove to open or close the window, thus creating the risk that the person might knock over a pot or lean into the flame of burner on the stove; (2) curtains in the window might catch fire; (3) the windows might get greasy; (4) someone cooking at a stove might get distracted by looking out the window. CRACK, however, did not display this rationale for users; it only displayed a brief critiquing message.

CRACK was meant as an improvement over an expert system approach in the sense that it empowered users by both providing expert advice but allowing those users to ignore this advice when they chose. The problem with CRACK was that, although it presented advice, it did not present the rationale behind that advice. Users were thus often uncertain about whether to follow the advice and how to act if they chose not to follow the advice. This deficiency was remedied by creating a new system, called JANUS, that combined the CRACK functionality with hypertext functionality that displayed the rationale for each critique using PHI. The new system had two fundamentally different kinds of functionality: support for constructing

designs (using construction kits) and support for design rationale. It was therefore named JANUS, after the Roman god with two faces.

JANUS presented the rationale for critiques in the form of domain-oriented issue base (DOIB) structured using PHI. These are collections of issues, positions, arguments, and subissues that commonly arise in a particular design domain. DOIBs had been developed since the late 1970s for a variety of domains including the design of residences, lunar and Martian habitats, neighborhood shopping areas, health care policy, and information retrieval systems. The JANUS system's DOIB provided issue-based information that was relevant to a wide range of kitchen design projects.

JANUS was successful not only in further empowering its users; it also answered the crucial questions raised in the PHIDIAS project about how and when to integrate support for rationale with support for CAD. After the JANUS system was implemented and judged successful, it was realized that these successes were actually implied by Schön's theory of Reflective Practice (Schön 1983).

Schön had viewed design as consisting of a repeated alternation between two processes, that he labeled Knowing-in-Action and Reflection-in-Action. Knowing-in-Action is the process of intuitively creating the form of a design—e.g., using pencils or CAD systems. It is a nonreflective process of unselfconscious engagement in the task of forming the design. This process continues until there is a *breakdown* of intuition when something unexpected happens. In conventional design, breakdowns correspond to the designer realizing that something is wrong with the design or that some unforeseen opportunity has arisen for improving the design. Once a breakdown has occurred, the designer changes to the mental process Schön calls Reflection-in-Action. This consists of reflecting on how to deal with the breakdown situation. This is a process of critical thinking in which the reasoning behind the design becomes explicit and it cannot be done simultaneously with the intuitive process of Knowing-in-Action. Once the designer has determined how to deal with the breakdown, Knowing-in-Action takes over again and implements the solution to the breakdown.

The JANUS group saw the intuitive construction of designs using construction kits as a clear example of Knowing-in-Action. Critiques corresponded to potential breakdowns. The PHI-based presentation of rationale for critiques provided support for the designers' Reflection-in-Action.

The JANUS functionality was integrated into PHIDIAS and then additional functionality was added. JANUS's hypertext functionality was implemented using the Document Examiner, which supported display of rationale but provided no support for authoring. Because PHIDIAS was based on MIKROPLIS, it also supported authoring of rationale, thus enabling designers to add their rationale to the DOIB used by the system.

This authoring of rationale was accomplished using a prototyping mechanism that enabled creation of a virtual copy of the DOIB. This enabled designers to add there rationale to the DOIB and even edit the DOIB without actually altering the original DOIB itself.

PHIDIAS also expanded the kind of knowledge-based critiquing available. In addition to critics that fired when designers positioned construction kit building blocks in the model of the designed artifact, PHIDIAS provided critics and rationale for the selection of building blocks from alternatives. PHIDIAS also provided knowledge-based agents that alerted members of design teams to potential conflicts between their work and the work of other designers in the team (McCall and Johnson 1997). PHIDIAS was applied to a variety of design domains, including the layout of computer networks in buildings, the design of lunar habitats and, of course, kitchen design.

4.2.2 Discussion

Critiquing is the most prominent feature of JANUS and PHIDIAS, but it is not the most important in its implications for rationale research in SE. The most important is the theory of Reflective Practice that these systems support. A central tenet of this theory is that it is a mistake to attempt to explicitly record the rationale for the process of Knowing-in-Action. This means that the traditional approach to rationale capture cannot be made to work for this part of the design process. The reason for this, according to Schön, is that forcing humans to make the reasoning behind Knowing-in-Action explicit would prevent Knowing-in-Action from taking place. But, if Knowing-in-Action cannot happen, then neither can design, at least according to Schön. The significance of this claim is that, if true, (1) it would go a long way towards explaining why it has proved so difficult to capture design rationale, and (2) it would imply that the traditional approach to the capture of design rationale can only succeed in capturing *part* of the reasoning that goes into decision-making in design. This does not mean, however, that capture is not possible, merely that it is not possible if one asks the person engaging in Knowing-in-Action to record the rationale. Such capture might effectively be accomplished by another person or by automated means such as those used by Myers et al. and described below.

One important contribution of critiquing is that it alerts decision-makers to the existence of rationale for a decision task without their having to ask whether it exists. This is a valuable contribution that makes it less likely that decision-makers will miss valuable information. However, critiquing is not the only mechanism that can do this. PHIDIAS also employs other

mechanisms that detect what task designers are engaged in and alert the designer to rationale for this task, thus implementing a general sort of task-based indexing of rationale in addition to its critiquing. Burge has implemented mechanisms in the Eclipse IDE that alert implementers to the existence of rationale relevant to particular pieces of code that they look at. This is somewhat similar to the task-based indexing in PHIDIAS, but there is the question of whether Burge's approach could be extended to include more general task-based indexing for SE.

One feature of both PHIDIAS and JANUS appears to have straightforward application to every activity of SE: the use of Domain-Oriented Issue Bases (DOIBs). Because any decision task can be represented as an issue, DOIBs would seem to be applicable to decision tasks of all types, including those for requirements determination, design, construction, testing, and maintenance.

Adapting the critiquing of JANUS and PHIDIAS to SE support systems presents an interesting challenge. This sort of critiquing is heavily dependent on CAD systems' use of *iconic models*. Iconic models are graphical models in 2D or 3D Euclidean space of artifacts that occupy 3D Euclidean space. In iconic models there is a natural correspondence—or *natural mapping*—between parts of the model and the parts of the real-world object it represents. In addition, the placement of a single element into an iconic model implies the existence of a whole battery of relationships with other elements in the model. These relationships include distance between elements, whether they are lined up vertically, horizontally or at an angle, whether they are collinear—and so forth. All the critics in JANUS and PHIDIAS are based on these implied relationships.

The only place that SE uses iconic models is in the design of graphical user interfaces (GUIs). This is therefore the one area where the approaches used in JANUS and PHIDIAS—as well as other systems described in this chapter—might find direct application to software projects.

Software designers generally create and use *symbolic models* rather than iconic models. By definition, the denotation relationships between elements of a symbolic model and the artifact it represents are arbitrary social conventions. Symbolic models can, however, come to feel like they also have a natural mapping if the relationships between symbols and real objects are truly standard, i.e., something universally accepted within a large group of people. The more software designers use models with standardized semantic meaning, the more natural this mapping will seem.

An open research question is whether the sort of rich collection of implied relationships found in iconic models can also be found in symbolic models. Since these models often take the form of graphs, it may well be that graphic theory might provide a way of deducing such relationships.

Perhaps conceptual schemas dealing with the types of elements and relationships amongst them could be used as the basis of critiquing in symbolic models. Whether a significant set of critics for SE can be developed remains to be demonstrated.

4.3 Automating the Capture of Design Rationale with CAD

4.3.1 The Rationale Capture Problem

This book emphasizes repeatedly that the biggest challenge facing the use of rationale in real-world projects is *the rationale capture problem*. This is the fact that it is extremely difficult to capture rationale in a real-world setting. The hallmark of this problem is that those involved in design and other SE activities often seem reluctant to record their rationale. Why this should be and what to do about it are controversial questions in current rationale research in SE as well as in other fields where rationale research is done.

Researchers in increasing numbers have come to the conclusion that the capture problem results from the intrusive and time-consuming nature of the traditional approach to rationale capture. In this approach, rationale must be structured according a given schema, such as IBIS, DRL or QOC, in order to be recorded. In other words, the initial recording of the rationale is in a structured form. There is little debate about the fact that this structuring process is labor intensive, but some claim that it is also disruptive to the free flow of intuitive and creativity thought in problem solving. Marshall and Shipman see all mandatory structuring as inhibiting user input (Marshall and Shipman 1999), and Fischer and his colleagues use Schön's theory of Reflective Practice to argue that the explicitly structured reflection interferes with the intuitive problem-solving process that Schön calls Knowing-in-Action.

On the other side of the debate are those who acknowledge that the capture process is intrusive and labor intensive but argue that it is worth it because of the benefits from having captured rationale and even from the process of structuring it. In the latter case, advocates of the traditional approach claim that the structuring process helps artifact developers to improve the consistency and thoroughness of their reasoning.

4.3.2 Solution Approach: Automating the Capture of Rationale

Myers, Zumel, and Garcia have done research on rationale for the design of physical artifacts (Myers et al. 1999), and they are among those who see the traditional approach as the central cause of the capture problem. Their strategy for solving this problem is to automate rationale capture to the greatest extent possible. In other words, they seek to use automated computer methods to capture rationale in a manner that is so unobtrusive that a designer can be completely unaware that capture is taking place. More specifically, they adopt the *generative paradigm* of Gruber and Russell (1996) and attempt to derive rationale from data obtained during design. Interestingly, they do not use the argument for this approach given by Gruber and Russell, which is that it is not possible during design to predict what rationale will be needed later. Instead, they use the argument that the unobtrusiveness of the approach is the decisive factor.

Myers and her collaborators adopt the approach of first recording the behavior of designers using a CAD system and secondly inferring from these records both a *design history* and *design intent*. *A design history* is an account of *what* designers did and *when* they did it; *design intent* is *why* they did what they did. The goal here is not to automate all rationale capture, but instead to automate capture of "important but low-level aspects of the design process," so that designers can limit their documentation efforts to the higher-level, "creative and unusual aspects" (Myers et al. 1999). The central insight on which their approach is based is that CAD systems often enable designers to perform operations on artifacts that are semantically meaningful in the application domain.

To derive a design history, they capture records of the atomic actions possible with the CAD system and then attempt to infer designers' behavior at higher levels of abstraction (lower levels of granularity). They derive a hierarchical account of designer behavior in terms of episodes created by grouping atomic actions. They also characterize the artifact in hierarchical terms as assemblies, subassemblies, and other groupings of parts. From these hierarchies of behavior and artifact structure they deduce *what decision tasks designers are undertaking and what decision alternatives they are exploring*. These decision tasks all have to do with determining features of the artifact; so they correspond to *questions* in QOC rather than the more general concept of *issues* in IBIS. The decision alternatives thus correspond to QOC *options*. It should be noted, however, that the analyses of Myers et al. make no reference to QOC or any other rationale schema.

To derive design intent, they use artificial intelligence (AI) techniques that speculate on user motives using so-called *design metaphors* and a

formally stated set of *requirements* for the artifact. Design Metaphors are sequences of designer activities that suggest explanations for these activities.

4.3.3 Implementation: The Rationale Construction Framework

Myers, Zumel, and Garcia created a software system called the Rationale Construction Framework (RCF) to implement and test their ideas about automatic capture of design rationale. RCF has three main components:

- An enhanced CAD tool
- A Monitoring module
- A Rationale Generation module (RGM)

The CAD tool used was the commercially available MicroStation95, which had capabilities for modeling in the domain of electromechanical design, in which the ideas for automated rationale capture were to be tested. This tool was enhanced to enable designers to indicate the semantic type of graphical objects together with type-specific semantic attributes. For example, a given graphical object might be assigned the semantic type *gear* and given gear-specific attributes such as *number of teeth* and *gear ratio*. A second augmentation of the CAD tool added a set of analysis programs linked directly to objects in the CAD drawing. A third augmentation added the ability for designers to select graphical objects from a predefined library of semantically meaningful graphical objects.

The Monitoring module in RCF unobtrusively tracks the operations of the designer with the CAD system. Those operations that are relevant to design rationale are then passed on to the RGM in real time. Such operations include the creation, deletion, and modification of design objects, the selection of such object from the library and their use as parts of other objects, as well as the assignment of semantic information to objects. Undoing and redoing are also passed on to the RGM as is the use of analysis programs.

The RGM performs the majority of the inference done by the RCF system. It constructs *a symbolic model of the artifact* being designed. It then uses this model and the *design event log* received from the Monitoring module together with a formally specified set of *design requirements* and the *design metaphors* to construct the design rationale.

To derive *design intent*, the RGM focuses on explaining the changes to the artifact model during design. Design metaphors play a major role in

explaining these changes. Two examples of such metaphors are *refinement* and *part substitution*. Other metaphors help to identify important relationships between object that are not formally indicated in the model. Such metaphors can detect when objects are created, modified, and deleted together.

Identification of relationships between design requirements and the changes in design objects also plays a crucial role in deriving design intent. Specifically, RCF constructs hypotheses that such changes are attempts to satisfy requirements. Such hypotheses can be constructed with or without domain-specific background knowledge, though the latter provides richer accounts of design intent. Once hypotheses are constructed, they can then be supported or undermined by further evidence collected as the design effort proceeds.

Myers, Zumel, and Garcia (1999) describe the testing of RCF in a project aimed at designing a three-degree-of-freedom surgical robot arm. The system recorded and analyzed design activities from initial design through multiple stages of refinement. RCF was successful in describing designer activities at several levels of abstraction, identifying stages where the designer concentrated on revisions of particular parts or subassemblies, identifying the results of design tradeoffs, and in explaining key changes in the design.

4.3.4 Discussion

The rationale capture problem is of such importance for the future of rationale usage that a claim to capture a significant portion of it automatically cannot be ignored. The work on the RCF looks like a promising extension of research on domain-oriented design environments. Myers et al. have used the same sort of semantically meaningful components found in the construction kits of JANUS and PHIDIAS, information used by those systems to identify design decision tasks, decision alternatives and decisions taken. However, RCF's abilities to identify and characterize designer activities and to speculate on the reasons for them goes far beyond what JANUS and PHIDIAS can offer. The RCF approach provides a way of capturing rationale for the intuitive Knowing-in-Action that Schön claims is disrupted by the explicit reflection that traditional rationale capture requires.

RCF, like JANUS and PHIDIAS, relies on the natural semantic mapping available in the iconic models that CAD systems create. This means that there is a question about how well the RCF approach would transfer to the purely symbolic models that are used in software design. However, to the

degree that the symbols used in software design models are genuine standards and not the arbitrary creations of individual designers, transfer would seem to be possible.

If transfer of the principles of RCF to software design is possible, the benefits would be considerable. Of prime importance, of course, is that it might solve at least part of the capture problem. But in addition, RCF's emphasis on rationale for *explaining changes* has crucial implications for change analysis as well as the iterative and evolutionary methods of software development.

4.4 Parameter Dependency Networks as Design Rationale

4.4.1 The DRIVE System and Parameter Dependency Networks

de la Garza and Alcantara (1997) describe a software system, called Design Rationale in Value Engineering (DRIVE), that provides additional computer support to aid designers who document their rationale. As is often the case, the additional computer support requires a higher level of formalization of rationale than is common with most rationale management approaches. The DRIVE approach, however, can be viewed as a simple extension of the formalization required for Design Space Analysis in QOC.

The DRIVE system enables designers of physical artifacts to create dependency relationships between the parameters of objects found in a model of a physical artifact that is being designed. Such dependencies can then be used as rationale for design decisions made using a CAD subsystem. More specifically, these dependencies constitute rules—or more accurately, rules of thumb—for design decisions. These rules can then be used to critique the decisions that the designer makes using CAD. The DRIVE system uses these rules to detect conflicts created by decisions and then alerts the designer to the existence of the conflicts as they use the CAD subsystem. The designer can then either resolve the conflicts immediately or postpone their resolution. Conflict resolution is accomplished by altering the design, altering the dependency rule or canceling the dependency rule for a specific CAD decision.

There are two types of parameter dependencies that DRIVE supports. One type is the dependency of the value of a parameter (attribute) of an object on the value of a parameter of an object, where either the parameters are different or the objects are different or both. The second

type is a dependency of a parameter constraint on the values of other parameters. There are also two ways in which dependencies can be represented: as mathematical formula or as an if–then rule. The following is an example of an if–then dependency of a parameter constraint on a parameter value as it would be expressed in the DRIVE system (de la Garza and Alcantara 1997):

If [Mechanical Room]:[General Function]
  Is equal to "House Mechanical Equipment"
Then [Mechanical Room]:[Fire Resistance Rating]
  (minimum value) is not less than 2 hours

In ordinary language this rule says that, if the general function of a "Mechanical Room" is to house mechanical equipment, then this room should have a fire resistance rating of at least 2 hours. It should be noted that in DRIVE each such rule is accompanied by natural language text that explains the rule and can provide additional arguments for them.

4.4.2 Discussion

4.4.2.1 How DRIVE's Parameter Dependency Networks Relate to Other Approaches to Rationale

DRIVE's treatment of rationale resembles QOC's Design Space Analysis in the sense that it deals only with rationale for features of the designed artifact. However, DRIVE's description of artifact features is more specific than QOC's. QOC only provides a textual description of a feature, but DRIVE provides a three-part feature description: (1) a type of object, (2) a parameter (i.e., an attribute) of the object, and (3) one or more allowed values of that parameter. While QOC, like DRL, evaluates a proposed artifact feature by means of assessments with respect to criteria, DRIVE assesses a proposed decision about a parameter value by means of other parameter values.

The DRIVE system resembles both JANUS and PHIDIAS in its use of a critiquing system that delivers textual rationale to designers of physical artifacts as they work in a CAD system. The crucial innovations of DRIVE are (1) the use of parameter dependency networks as the basis for critiquing and (2) enabling designers to create their own critiquing rules.

The dependency relationships used by de la Garza and Alcantara in DRIVE are more specific that the dependency relationships used by Burge in RATSpeak and her SEURAT software. Burge's dependencies are

natural language arguments that do not enable the computation of values and constraints as in DRIVE, and of course, DRIVE's parameter dependency networks are far more specific than the dependency network between decisions (issues) that PHI uses to structure rationale.

As with JANUS, PHIDIAS, and the Rationale Capture Framework of Myers et al., DRIVE depends on the natural association of semantic meaning with the graphical objects used in the CAD system, i.e., the natural mapping of iconic models. This is crucial because the critiquing depends on the rules applying to classes of objects. In DRIVE, this is, in effect, accomplished using *is-a* and *has-a* relationships, though the actual implementation of these concepts is domain dependent and complex.

4.4.2.2 Significance for Software Engineering Rationale

DRIVE's use of algebraic formulas for dependencies seems unlikely to find extensive application in SE, but its if–then dependencies would seem to have a wide range of applications in SE. They constitute a more specific and more computable version of the argumentative dependencies between decision alternatives found in RATSpeak. This is especially significant in view of the fact that RATSpeak was created in an attempt to tailor DRL to the needs of software engineers who do maintenance. The if–then computational dependencies used in DRIVE are especially promising for change analysis, which is one of the most important and popular uses of rationale in SE. Investigating the potential value of parameter dependency networks should therefore be an important topic for future research on software engineering rationale.

4.5 Case-Based Reasoning as Design Rationale

4.5.1. From Automated Case-Based Reasoning to Case-Based Design Aids

Case-Based Reasoning (CBR) (Riesbeck and Schank 1989; Kolodner 1993) began as a branch of artificial intelligence (AI) research. It was meant as an alternative to the dominant AI approach, sometimes called Model-Based Reasoning (MBR). MBR had run into well-known difficulties, and CBR researchers thought their approach offered a way around many of these difficulties. MBR is about reasoning from principles, often in the form of *rules* or *productions*. Roughly speaking, CBR does not

reason from principles but from similarity of a current problem to cases of previously solved problems.

CBR originally dealt with the creation of automated systems that mimicked the human ability to use knowledge of prior cases to deal with new kinds of problems, but it eventually became clear that the number and complexity of cases it could deal with in a completely automated manner was quite limited (Narayanan and Kolodner 1995). To address these problems, Kolodner began to look at developing nonautomated CBR systems that aided human problem solvers in complex problem domains. The idea was that, by learning how to aid humans who solved complex problems, CBR researchers would get better insights about how these problem solvers use large collections of complex and often incomplete cases. These insights could ultimately be used to improve fully automated CBR systems.

The primary applications domain chosen for study was architectural design, i.e., the design of buildings. Kolodner and her computer-science colleagues at Georgia Tech worked with faculty and students in the Department of Architecture at that institution to create Case-Based Design Aids (CBDAs) and populate them with information about buildings. This effort resulted in a number of systems, including two versions of the ARCHIE CBDA for building design and DesignMuse, a generalized authoring tool for creating CBDAs for different domains of physical artifact design. Originally the building domain was restricted to the design of courthouses, but it was later expanded to deal with libraries and tall buildings.

CBDAs are case libraries for design, i.e., "structured, indexed and searchable databases of analyzed case studies" (Narayanan and Kolodner 1995) containing descriptions and evaluations of existing designs, e.g. the designs of existing buildings. The descriptions are typically represented using multiple media, including text and graphics. The purpose of a CBDA is to provide information about lessons learned from the experiences of previous designs so that current designers can avoid the pitfalls of past projects and benefit from solution ideas that have proved successful in such projects.

CBDAs contain cases structured around four major categories of information: *descriptions, problems*, *stories*, and *responses*. *Descriptions* are multimedia representations of designed physical artifacts. In ARCHIE these take the form of annotated CAD drawings of floor plans, elevations, and sections of buildings, as well as sketches, photographs, and animations.

Problems are descriptions of unresolved conflicts that are common and persistent in a type of building. The following is an example of a problem statement:

> Clerestories [narrow, horizontal bands of windows just beneath a ceiling] and skylights can help light large interior spaces, but they can also cause costly environmental problems. They can create hot spots in warm weather and increase air-conditioning costs (Zimring et al. 1995).

Stories are brief representations, in text and other media, of how the problem or a solution has manifested itself in a particular building. The following is an example of a story about a solution for the above-given problem:

> In the Gwinnett County Courthouse clerestories and skylights were used to illuminate the interior atriums. The high, angled skylights are made of tinted glass. The depth and tinting of the skylights helps prevent direct sunlight from flooding the building (Zimring et al. 1995).

There can be many stories for a given problem. The same is true for problems and responses.

Responses are general strategies a designer might consider for resolving the problems. There can be many suggested responses for each problem. The following is an example of a multipoint response to the above problem:

> Use tinted glass where possible. Use clerestories rather than skylights. Angle and inset skylights to block direct sun. Use electronically moveable/controllable louvers (Zimring et al. 1995).

There can be many such responses to a problem.

CBDAs enable designers to retrieve information either by using special case-based retrieval mechanisms or by browsing using hypertext links. One of the special retrieval mechanisms automatically retrieves relevant cases based on the designer's description of a current problem's goals and constraints. The other uses an induction algorithm that clusters cases to build a hierarchical index. Hypertext links in ARCHIE and other CBDAs connect design descriptions to stories, stories to problems, and problems to responses (Zimring et al. 1995.)

4.5.2 Discussion

4.5.2.1 Design Case Libraries as Design Rationale

The creators of ARCHIE and other CBDAs make no claim that the information in these systems is design rationale (DR); yet there seems to be little reason to doubt that it is. After all, the cased-based information in ARCHIE deals with design problems, design solutions, and solution strategies. It includes descriptions and evaluations of designed artifacts. Its sole function is to provide information that can help designers to make better decisions, i.e., to aid designers' reasoning. And, as with almost all other rationale approaches, the information in a CBDA is organized as a hyperdocument of links and nodes of text and graphics.

While case-based information about design clearly must be counted as design rationale, it differs profoundly from all other known types of design rationale hyperdocuments, including those based on IBIS, PHI, QOC, DRL, SCA or any of the SE-specific approaches currently in existence. CBDAs provide a fundamentally different perspective on how to go about collecting, structuring, indexing, retrieving, and using design rationale. And this new perspective comes with a solid intellectual pedigree in cognitive science and computer science. No picture of research on rationale would be complete if it omitted the work on CBDAs like ARCHIE. A crucial task for future rationale research will be to fit case-based design rationale into the overall landscape of rationale approaches.

4.5.2.2 Design Case Libraries as an Alternative Approach to Reuse of Rationale

In the rationale research literature there have been two main approaches to reuse of rationale. One is the addition of rationale to design patterns. The other is the use of domain-oriented issue bases (DOIBs). Design case libraries represent a third fundamental alternative.

One way to attempt to understand the crucial differences between the three alternative approaches to rationale reuse is to compare the ways in which they use generalization and specificity when reasoning about new projects. Rationale linked to patterns represents an attempt to create generalized stores of reasoning, in other words, collections of rationale that involve generalizations that apply to many specific design projects. This can be seen as reasoning from principles, the central notion of MBR in AI research. Case libraries for design, however, are based on a fundamentally different approach to reasoning, namely CBR, which involves reasoning

from cases of previous, specific projects to draw conclusions about a current, specific project.

DOIBs involves a type of reasoning that falls in between the MBR-type of reasoning of pattern-based rationale and the CBR reasoning of case-based design rationale. Where it sits in between MBR and CBR depends on which of two distinct modes a DOIB is used in. One mode attempts to create collection of texts—including issues, positions, and arguments—that can be reused *as is* in many projects. This mode is exemplified by the use of the DOIB for kitchen design in JANUS. In its reuse of unmodified information in many specific projects this mode is like pattern-based rationale except that there is no claim of either completeness or correctness for the texts in the DOIB.

A second mode of use of DOIBs is provided by the virtual copying of hypermedia networks that is available in PHIDIAS. This enables the creation of a new DOIB by making and modifying a virtual copy of the original DOIB using the prototyping inheritance mechanism in PHIDIAS. This is typically used to create a more specific DOIB than the original, in particular, one tailored to a specific project. This mode of DOIB usage is in between the general-to-specific reasoning of pattern-based rationale and the specific-to-specific reasoning of case-based design rationale, because it uses generalized information but adapts it to a particular project.

There is also a third way in which the hypermedia network inheritance functionality of PHIDIAS can be used. In this approach a new project-specific issue base is created by virtually copying and modifying a previous project-specific issue base. This approach takes a significant further step towards the type of reasoning used in cased-based design rationale, but the schema for issue-based rationale remains dramatically different from the schema for cased-based rationale of CBDAs like ARCHIE.

4.5.2.3 The Relevance of Case-Based Design Rationale to Software Engineering

Despite the fact that CBDAs have been created only for the domain of physical artifact design, there seems to be no fundamental reason why they could not be applied to software design and perhaps even to the full spectrum of development and maintenance activities in SE. Given the fact the case-based approach to rationale is so fundamentally different from other rationale approaches, exploring its potential for SE would seem to be an important topic for research in software engineering rationale.

Where case-based design rationale would appear to have special promise is in the design of human–computer interaction (HCI), because it is fundamentally a user-centric, rather than decision-centric, approach to

rationale. Currently, there is only one user-centered approach to rationale that is usable for this purpose, namely Scenario-Claims Analysis (SCA). In fact, the current heavy emphasis on both static and animated graphical representation of artifacts in CBDAs would be directly applicable to a case library of HCI designs. Such a case-based approach to interface design might be a useful complement to SCA, though it also seems possible that the two approaches might be integrated.

Of course, most of SE does not deal with the creation of an intrinsically graphical artifact as is the case with both physical artifact design and HCI design. However, software design, like the design of physical artifacts, does involve the use of models that have a graphical representation. Such models can be annotated and could easily have *problems, stories,* and *responses* associated with them. While these models are purely symbolic in nature and do not have the *natural mapping* to the artifacts they represent that *iconic* models like floor plans have to buildings, this does not seem to constitute an insurmountable obstacle to the creation of CBDAs for SE.

4.6 Summary and Conclusions

There are fundamental issues to be resolved before much of the research on rationale in domains of physical artifact design can be applied to the design of software; but the ideas in this research are important enough that the effort to resolve these issues seems worthwhile. Above all, it is the value of this work in the areas of *rationale capture* and *change analysis* that recommends it to software engineers. It seems ironic that the work on change analysis has made such progress in physical artifact design, where there is generally much less change—especially change due to iteration and evolutionary development—than is characteristic of software design. It seems appropriate that software engineers endeavor to learn and benefit from this progress.

Finally, it is interesting to note that all of the projects described in this chapter in some way apply insights from artificial intelligence (AI) research to the support of rationale. In particular, all but one of these projects—the one based on Case-Based Reasoning—bring active computational aids to support the capture and retrieval of rationale in artifact creation. This suggests that researchers in SE should seek to answer the questions of what other ideas from AI and what other computational aids might support rationale not only in software design but in the full spectrum of SE activities.