Worcester Polytechnic Institute
Digital WPI

Computer Science Faculty Publications | Department of Computer Science

1-2002

Discovering a Research Agenda for Using Design Rationale in Software Maintenance

Janet Burge
Worcester Polytechnic Institute, jburge@cs.wpi.edu

David Brown
Worcester Polytechnic Institute, dcb@wpi.edu

Follow this and additional works at: https://digitalcommons.wpi.edu/computerscience-pubs

Part of the Computer Sciences Commons

Suggested Citation
Burge, Janet , Brown, David (2002). Discovering a Research Agenda for Using Design Rationale in Software Maintenance. .
Retrieved from: https://digitalcommons.wpi.edu/computerscience-pubs/120

This Other is brought to you for free and open access by the Department of Computer Science at Digital WPI. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

DISCOVERING A RESEARCH AGENDA FOR USING DESIGN RATIONALE IN SOFTWARE MAINTENANCE

J. BURGE, D. C. BROWN
AI in Design Research Group
Department of Computer Science
WPI, 100 Institute Road
Worcester, MA 01609, USA

Abstract. Design Rationale consists of the reasons behind decisions made while designing. This information would be particularly useful during software maintenance. In this paper, we describe a study performed to investigate the content, structure, and use of design rationale during maintenance. The major goal of this study was to discover an agenda for further research into the use of design rationale for software maintenance.

1. Introduction

For a number of years, members of the Artificial Intelligence (AI) in Design community have studied *Design Rationale* (DR), the reasons behind decisions made while designing. Standard design documentation consists of a description of the final design itself: effectively a "snapshot" of the final decisions. Design rationale offers more: not only the decisions, but also the reasons behind each decision, including its justification, other alternatives considered, and argumentation leading to the decision (Lee, 1997). This additional information offers a richer view of both the product and the decision-making process by providing the designer's intent behind the decision. DR is invaluable as an aid for revising, maintaining, documenting, evaluating, and learning the design.

Rationale for past decisions is especially useful during software maintenance. One reason for this is that the software lifecycle is a long one. Large projects may take years to complete and spend even more time out in the field being used (and maintained). Maintenance costs can be more than 40 percent of the cost of developing the software in the first place (Brooks, 1995). The panic over the "Y2K bug" highlighted the fact that software systems often live on much longer than the original developers intended. Also, the combination of a long life-cycle and the typically high personnel turnover in the software industry increases the probability that the original designer is unlikely to be available for consultation when problems arise.

1.1 DIFFICULTIES WITH RATIONALE

While rationale has great potential value, rationale is not in widespread use. One difficulty, despite a good deal of research, is the capture of design rationale. Recording all decisions made, as well as those rejected, can be time consuming and expensive. The more intrusive the capture process, the more designer resistance will be encountered.

Documenting the decisions can impede the design process if decision recording is viewed as a separate process from constructing the artifact (Fischer, et. al., 1995). Designers are reluctant to take the time to document the decisions they did not take, or took and then rejected (Conklin and Burgess-Yakemovic, 1995). A real danger is the risk that the overhead of capturing the rationale may impact the project schedule enough to make the difference between a project that meets its deadlines and is completed, versus one where the failure to meet deadlines results in cancellation (Grudin, 1995).

1.2 USES OF RATIONALE

The key to making the capture worthwhile, as well as providing requirements for DR representation, is the use for, *and usefulness of,* the rationale. There are a number of potential uses for DR. These include:

- *Design verification* – using rationale to verify that the design meets the requirements and the designer’s intent.
- *Design evaluation* – using rationale to evaluate (partial) designs and design choices relative to one another to detect inconsistencies.
- *Design maintenance* – using rationale to locate sources of design problems, to indicate where changes need to be made in order to modify the design, and to ensure that rejected options are not inadvertently re-implemented.
- *Design reuse* – using rationale to determine which portions of the design can be reused and, in some cases, suggest where and how it should be modified to meet a new set of requirements.
- *Design teaching* – using rationale to teach new personnel about the design.
- *Design communication* – using rationale to communicate the reasons for decisions to other members of the design team.

- *Design assistance* – using rationale to clarify discussion, check impact of design modifications, perform consistency checking and assist in conflict mitigation by looking for constraint violations between multiple designers.
- *Design documentation* – using rationale to document the design by offering a picture of the history of the design and reasons for the design choices as well as a view of the final product.

Because *use* is the key behind the value of the rationale, the focus of our investigation is on how rationale can be *used* to assist in software maintenance.

In this paper, we describe a study performed to discover a research agenda for further work in using design rationale for software maintenance. This study investigated the content, structure, and use of design rationale. The following sections discuss observations made during this study and provide the resulting research agenda.

This paper is structured as follows: in section 2, we describe related work. In section 3, we describe the overall goal of our study. Section 4 describes the study and section 5 presents the results. Section 6 outlines our research agenda that resulted from the study and section 7 gives the summary and conclusions.

2. Related Work

Design Rationale is a kind of Knowledge Representation. How the DR can be used depends on its representation format and content (Lee, 1997). Design Rationale representations vary from informal representations such as audio or video tapes, or transcripts, to formal representations such as rules embedded in an expert system (Conklin and Burgess-Yakemovic, 1995). A compromise is to store information in a semi-formal representation that provides some computation power but is still understandable by the human providing or using the information.

Semi-formal representations are often used to represent argumentation. Argumentation notations provide a structure to indicate what decisions were made (or not made) and the reasons for and against them. Some examples are Questions, Options, and Criteria (QOC) (MacLean, et. al., 1995), Issue Based Information Systems (IBIS) (Conklin and Burgess-Yakemovic, 1995), and DRL (Decision Representation Language) (Lee, 1990).

There are also many different ways to capture DR. One approach is to build the rationale capture into a system used for the design task. An example is RCF (Rationale Construction Framework) (Myers, et. al., 1999), which integrates DR capture into an existing design tool.

DR has a variety of uses. Systems such as JANUS (Fischer, et. al., 1995), critique the design and provide the designers with rationale to support the criticism. Others, such as SYBIL (Lee, 1990), verify the design by checking that the rationale behind the decisions is complete. C-Re-CS (Klein, 1997) performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. InfoRat (Burge and Brown, 2000) performs inferencing over the rationale to verify that the rationale is complete and consistent, and to also evaluate that decisions made were well supported.

There has also been work on using design rationale in software design. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale (Pena-Mora and Vadhavkar, 1996). Co-MoKit (Dellen, et. al., 1996) uses a software process model to obtain design decisions and causal dependencies between them. WinWin (Boehm and Bose, 1994) aims at coordinating decision-making activities made by various “stakeholders” in the software development process. Bose (Bose, 1995) defined an ontology for the decision rationale needed to maintain the decision structure. The goal was to model the decision rationale in order to support decision maintenance by allowing the system to determine the impact of a change and propagate modification effects.

Less work has been done to study the usefulness of DR. Field trials were done using itIBIS and gIBIS for software development at NCR (Conklin and Burgess-Yakemovic, 1995). Capturing rationale was found to be useful during both requirements analysis and design. In particular, several errors were found during design that would not have been uncovered until much later when the code was written. IBIS also helped with team communication by making meetings more productive. A study was also performed using DR documents to evaluate a design (Karsenty, 1996). In this study, 50% of the designers’ questions were about the rationale behind the design and 41% of those questions were answered using the recorded rationale.

3. Goal

Our design study had a number of goals. First, we wanted to collect some software design rationale to obtain a better understanding of what software design rationale was. Second, we wanted to begin investigation into how rationale was affected and used during software maintenance. Third, we hoped to produce an agenda for further research into using design rationale during software maintenance. Our study, described in section 4, resulted in initial insight into the first two goals and provided us with a research agenda with which to continue our investigation.

3.1 SOFTWARE DESIGN RATIONALE

Design rationale could be generated at any stage of the software design process. Figure 1 shows the development phases and the rationale that could be generated during each of them.

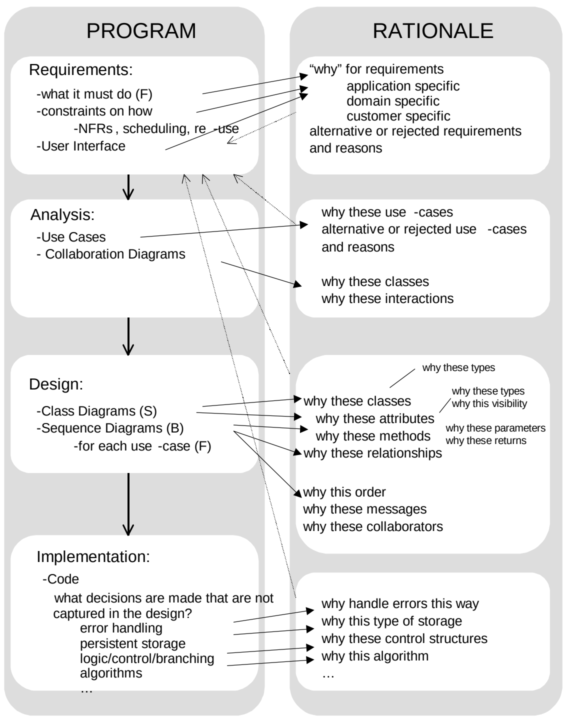

Figure 1: Software Development Phases and Rationale

This rationale could describe many different types of decisions:

- *Requirements* – rationale could exist for the existing requirements and for requirements that were considered but then rejected. There would be rationale for the user interface design if the design was performed during the requirements phase.
- *Analysis* – rationale could be associated with use-cases and with the partitioning of the problem into analysis classes and collaboration diagrams.
- *Design* – rationale could be associated with any portion of any design artifact. This could include reasons behind the choice of the design classes, the attributes (including reasons for data types and visibility), the methods, etc.

- *Implementation* – rationale could describe the choice of algorithms, data structures, persistent storage, and more.
- *Maintenance* – rationale could describe both why the modifications were necessary, as well as the reasons behind the design and implementation choices necessary for the modification.

Capturing all this information would present a significant amount of overhead to the software developer. During our study, we collected data on the decisions made and the rationale collected as a start at better understanding what information comprised design rationale for software.

3.2 SOFTWARE MAINTENANCE

In order to investigate using DR during software maintenance, we first must look at the maintenance task itself. There are a number of different classifications for types of software maintenance (Chapin, 2000). We looked at three types in this experiment: corrective, perfective, and enhancive. We used an existing meeting scheduler system to investigate the different types.

1. *Corrective* – Corrective maintenance involves correcting failures of the system (Lientz and Swanson, 1980). In the meeting scheduler, there was a minor bug where meetings could not be cancelled after saving the schedule if the time period indicated exactly overlapped the meeting duration.
2. *Perfective* – Perfective maintenance involves "perfecting the system," improving processing, performance, or maintainability (Lientz and Swanson, 1980). The meeting scheduler will not allow users to schedule two meetings that overlap. The initial version of the system did not check for this until after prompting the user for the name of the meeting. An improvement was proposed to verify the validity of the time range before asking the user for more information. This change was put into the perfective category since it did not affect the result of the scheduling operation but improved the experience for the user.
3. *Enhancive* – Enhancive maintenance involves replacing, adding, or extending "customer-experienced functionality" (Chapin, 2000). The initial meeting scheduler system allowed the user to create a single meeting schedule. An enhancement was proposed that allowed the system to be used as a conference room scheduler where the user could select a room and then reserve a time slot for the meeting. This extended the original functionality by maintaining a meeting schedule for each conference room.

3.3 DISCOVERING A RESEARCH AGENDA

To drive and evaluate our research into using rationale for software maintenance, we will develop a system that supports the maintainer. This system will present the relevant DR when required and allow entry of new rationale for the modifications.

The new DR will then be verified against the existing DR to check for inconsistencies. There are several types of checks that should be made: structural checks to ensure that the rationale is complete, evaluation, to ensure that it is based on well-founded arguments, and comparison to rationale collected previously for similar changes to see if the same reasoning was used. In the latter, the previous rationale could be used as a guide in determining the rationale for the new change. The system will also propagate any necessary changes to the existing DR as well as alerting the maintainer if the code modifications are the same as those made earlier and then rejected.

Our research, and development of this system, will require examining at least the following questions:

a) What types of design rationale are present at the different phases of the software development process?
b) What is the relationship between rationale collected during the different phases?
c) Are there portions of the design or phases of the development process (Figure 1) where rationale capture would be more useful than others?
d) What is the appropriate level of detail to capture in the rationale that will be useful, yet minimize the collection burden on the user?
e) How do modifications to the software affect the rationale?
f) Does rationale differ for different types of software modifications?
g) How does rationale for modifications differ from rationale for the initial design?
h) How can rationale assist during software maintenance?

We hope to use the answers to questions a and b to assist us in answering c. We expect that the types of rationale present will be similar to those shown in Figure 1. The relationship between rationale at different phases may only be via the development artifacts produced as part of the design and implementation. We need to answer question d to determine the representation for our rationale. Questions e, f, and g investigate how rationale is affected during different types of software maintenance. Question h has a number of possible answers and drives this research.

4. Study Description

One of the difficulties in studying potential uses for software design rationale is that there are few (if any) examples of it available for analysis. In order to better understand software design rationale, its role in software maintenance (both as a product and an input), and to provide a research agenda for further investigation, we performed a small design study that looked at rationale for an initial design and at rationale that was generated/changed when modifications were performed. Modifications were examined because our main interest is in how rationale can be used to assist software maintenance.

Since the focus of our work is how DR can be used during software maintenance, an existing system, a Meeting Scheduler, was used. This system had the following useful properties:

- Requirements, use-cases, and source code were available;
- The system made use of a pre-existing component;
- The system had (at least) one error in the current implementation that was typical of the types of errors that would need to be repaired during maintenance.

The system studied was a meeting scheduler system written in Java. It used a previously developed component as part of its user interface that allowed the user to enter meeting information into a schedule. The following sections describe the artifacts and rationale created for the initial design and each of the proposed modifications.

4.1 INITIAL DESIGN

The system being modified had the following design artifacts available: requirements, use-cases, and source code. These were augmented by reverse-engineering the system to produce Unified Process (Jacobson, et. al, 1999) development artifacts focusing on parts of the system that were most likely to be affected by the proposed modifications. This involved creating user interface storyboards, collaboration diagrams, class diagrams, and event trace diagrams.

During this process, rationale was collected for decisions that involved conscious choices between multiple alternatives. The rationale format was kept simple in order to lessen the burden on the developer. Figure 2 shows the graphical convention used in documenting the rationale.

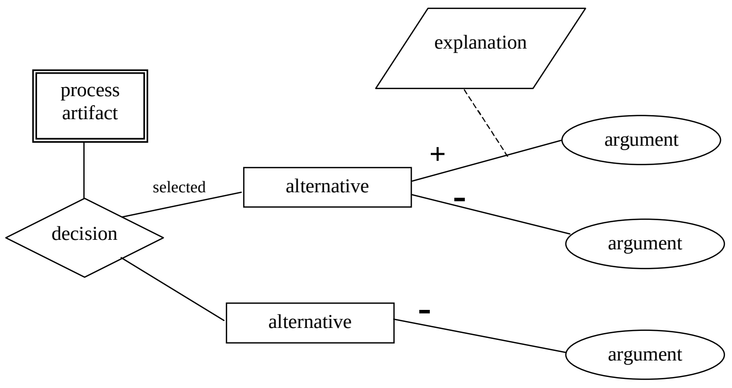

Figure 2: Rationale Components

This contained the following components:

- *Process artifact* – this could be a requirement, a display element, a use-case, a piece of code, or any portion of the system being developed.
- *Decision* – this is the decision that the rationale is documenting.
- *Alternatives* – these are the different alternatives considered to implement the decision.
- *Argument* – reasons for and against the alternatives (for marked with a “+” and against marked with a “-”).
- *Explanation* – the (optional) reason explaining why an argument applies to a particular alternative.

During each phase of the development process, the applicable Unified Process artifacts were created along with the rationale behind them:

- *Requirements Phase* – In most cases, the system is developed to meet a set of customer needs and desires that may not be fully explained. Requirements are developed to indicate what the system must do to satisfy these needs. There may be more than one way in which this can be done, hence the need to choose between alternative requirements and to provide reasons for the requirements chosen. Initial user interface design was also done during this phase.
- *Analysis Phase* – In the Analysis Phase, use-cases, analysis classes, and collaboration diagrams were developed. In the Unified Process, there are three types of analysis classes: boundary, control, and entity. Rationale was collected to indicate the reasons behind the type of class used, specifically the reasons for distinguishing between boundary and control classes.
- *Design Phase* – The Design Phase consisted of developing class diagrams and sequence diagrams. Rationale was collected to indicate

the reasons behind the choice of classes and allocation of responsibilities.

- *Implementation Phase* – The primary output of the Implementation Phase was the source code. Rationale was collected to indicate reasons behind the lower level design decisions made while writing the code. This included detailed information about data structures and algorithms.

4.2 CORRECTIVE MAINTENANCE – MINOR BUG IN THE PROGRAM

This exercise consisted of looking for a fairly minor error that occurred under a specific set of circumstances. The error turned out to be due to a misunderstanding on the part of the developer of how a particular Java method call worked. This was easily corrected by writing a new method that performed the desired function, rather than using an existing method that did not work as expected. The modification affected the design level, since a new method was added, and the code level, the implementation and use of the method. The rationale was updated to capture both the original decision and the alternative used to replace it.

4.3 PERFECTIVE MAINTENANCE – REVISITING THE DESIGN FOR USABILITY

In this case, a design decision from the original design was revisited to improve the usability of the scheduling system. Unlike the previous modification, this one started at the analysis level with the collaboration diagrams and then propagated down to the implementation.

4.4 ENHANCIVE MAINTENANCE – EXTENDING THE FUNCTIONALITY

This exercise involved extending the Meeting Scheduler system into one that scheduled meetings in different conference rooms. This was a significant increase in functionality since it involved saving several different schedules that could be moved between by selecting different conference rooms.

5. Results

The following sections describe what was learned during the initial design, the corrective maintenance modification, the perfective maintenance modification, and the enhancive maintenance modification.

5.1 INITIAL DESIGN

Rationale was generated during each phase of the development process. Some observations were specific to design phases while others apply to the rationale overall.

5.1.1 Phase Specific Observations

In the Requirements Phase, rationale consisted of the arguments for and against the candidate requirements as well as relationships between requirements. There are a number of different types of arguments. In some cases, the arguments capture a relationship between requirements and indicate which requirements cannot exist independently from each other. The argument may be that a candidate requirement supports a non-functional requirement (NFR) that is part of the base set of requirements (i.e., it is an NFR that directly supports a user request, such as a requirement to use a pre-existing component). In other cases, the arguments can be quality attributes that are not specifically mentioned as requirements but that are compelling reasons for preferring one alternative over another (where, in this phase, alternatives are in fact different requirements).

Much of the rationale captured during the Analysis Phase consisted of reasons for the categories (boundary, entity, or control) assigned to the analysis classes. This rationale is specific to the Unified Process since other software development methodologies do not use different types of classes during the analysis phase. Rationale was also collected to explain why some requirements were not given use-cases. Again, this is process-specific rationale.

Rationale captured during the Design Phase centered on the class diagrams, rather than the sequence diagrams. Many of the major sequencing decisions were made at the analysis level and were captured in the collaboration diagrams. The detailed sequencing of events represented at the design level seemed to obscure more than it revealed.

When the Meeting Scheduler system was implemented, the rationale collected made a dramatic leap in the level of detail. The explanations for why particular arguments applied to particular decisions became extremely detailed. Some decisions were fairly generic. For example, when choosing the type of data structure (such as hash table vs. vector), the different structures could have default rationale.

5.1.2 General Observations

There needs to be a way to represent arguments at different levels of abstraction. In some cases, the same argument was used for different alternatives but with different meanings. For example, two different user interface designs could both be considered to be usable but for different reasons or to a different degree (one design may have the best utilization of screen real estate while the other may minimize keystrokes). There are also many different types of arguments – some will map back to an NFR, others are based on assumptions or on preferences. Recording detailed arguments is most informative but makes it difficult to compare arguments when performing inferencing over the rationale. If an ontology of arguments existed, it could be used to capture detailed arguments yet still allow them to be compared at a higher level. For example, screen real estate and keystroke minimization arguments could be rolled up into an evaluation of usability.

One surprise was that in most cases (except at the requirement level), requirements were not used as arguments for or against alternatives. Instead, the requirements were the reasons that the decisions were necessary. Usually alternatives were not recorded in the rationale if they were clearly in violation of the requirement that spawned the decision. On the other hand, it is quite possible that an alternative chosen to meet one requirement may violate other requirements. It is very important to record requirement violations in the rationale.

The original, simplified format proposed for the rationale did not have an "explanation" component. The explanations were added because there was a need to explain why an argument applied to a particular situation. For this reason, explanations are attached to the relationship between the argument and the alternative, not to the arguments themselves. It would be desirable to make arguments specific enough that explanations would be less necessary. This is not easy – as the decisions became more specific, so did the reasons behind the alternatives. It became more and more difficult to come up with general names and categories for the arguments. Similarly, during the latter development phases, the explanations for the alternatives became very detailed – not something that could be reasoned over. There needs to be a way to break explanations down into more manageable pieces that can fit into an argument ontology and allow for comparisons. It would be useful if a system could be developed to help with this process.

The representation used in this study, with its simple +/- links for the arguments, was insufficient to express enough information to accurately document decisions. These need to be made more detailed, possibly using the InfoRat (Burge and Brown, 2000) format of amount and importance.

5.2 CORRECTIVE MAINTENANCE

In this maintenance example, an alternative selected during the initial design was rejected because it did not work. This raised a number of questions. First, there needs to be a way to specify in the rationale that an alternative was tried and failed. This needs to be more specific than simply giving a reason of “failed” as an argument against an alternative. The conditions under which the alternative failed and the reasons for failure also need to be specified. In some cases, the circumstances under which an alternative failed (or conversely, succeeded) may change. The rationale can be used to point out if decisions should be re-evaluated.

When modifications are made, both the rationale for the decisions made as part of implementing the change and the rationale for the reason the change was necessary need to be represented. This could be rolled into the reasons for rejecting previously selected alternatives but that would not be as explicit as linking the reason for the change to the decision affected.

An interesting rationale observation was that the rationale is not a flat structure, even within a development phase. Making a specific decision will spawn sub-decisions, with rationale at both levels. For example, the bug in the Meeting Scheduler was due to a decision to use a Java-provided Equals method to compare two date classes. This method did not do what was expected so the alternative was rejected and the alternative to create a custom comparison method was chosen. This choice then spawned a number of sub-decisions that concerned how to implement the new method.

It is not clear how multi-level rationale would affect inferencing over the rationale for decision evaluation. If the support for two alternatives is being compared, would rationale for the sub-decisions for those alternatives be used in this evaluation?

5.3 PERFECTIVE MAINTENANCE

This was a case where assigning more detailed information to the arguments (such as amount and importance) would have captured exactly why the alternative was selected. Was it necessary to change a decision because the preferences changed, thereby making the original choice sub-optimal, or was the original decision poorly thought out? This is an important distinction to make and was not captured by the original rationale.

If a detailed rationale representation involving amount and importance (how much the argument applies and how important the argument is) were available then the rationale would have been useful in pointing out that this change should be made. If the alternative chosen was rated as less desirable than others, this could be detected automatically by evaluating each alternative. If the importance assigned to an argument was inconsistent with that elsewhere in the system, this could be checked for as well. If external preferences changed, therefore affecting the importance of the various arguments, this could be used to re-evaluate each alternative and point out ones that are no longer the best choice.

5.4 ENHANCIVE MAINTENANCE

This modification involved adding two new requirements. The rationale recorded for the modification was used as the rationale for these new requirements. It did not look any different than any other rationale and the requirements did not look any different from the requirements that were already present. One thing that occurred during the Requirements Phase was that a requirement spawned additional requirements. In this example, a new requirement was added to state the new functionality and a second requirement was added to provide support for that functionality.

During the enhancement, some alternatives were chosen because they supported future enhancements. This needs to be clearly indicated in the rationale since often this results in choices that may appear to be less efficient in the current implementation. There were also cases where some code was "temporary", i.e. this code would need to be removed when the anticipated additional enhancements were made. This code needs to be clearly marked so that it can be removed or modified later. Rationale can help to point out places that will require modification. There were also some design decisions made based on assumptions. Again, rationale could be used to point out these places if the assumptions later prove to be untrue.

6. Research Agenda

This study highlighted a number of areas that need to be addressed in order for rationale to be useful during maintenance.

6.1 RATIONALE REPRESENTATION

The simplified representation used in the experiment quickly proved to be insufficient to support software maintenance. In some cases, the same arguments could apply to different alternatives, but to different degrees. This can be addressed by giving each argument an importance and amount relative to the alternative being considered. This method has been used in other systems including KBDS (Bañares-Alcantara, et. al., 1995) and InfoRat (Burge and Brown, 2000).

There also needs to be more investigation into how certain special-case arguments are handled. For example, requirement violations (which may be represented as arguments against alternatives) need to be given sufficient importance to ensure that the alternative is not chosen. Also, if an argument is only valid under certain circumstances (which may change over the life of the product), then this also needs to be represented. It is also important to indicate explicitly when an alternative has been tried and rejected and why.

The rationale for modifications to the software also needs to be captured. If the modifications start at the requirements level this can be captured as rationale for the new requirements. If not, there needs to be a way to tie together the changes to the rationale and give the overall reasons for them.

The rationale representation structure also needs to be examined. This study showed that rationale is not a flat structure – some decisions then spawn additional sub-decisions. This needs to be represented. This will also affect inferencing over the rationale, as the support for sub-decisions may need to be factored into calculations of support for the parent decisions.

6.2 ARGUMENT ONTOLOGY

In order to support inferencing, the arguments for and against alternatives need to be represented in a way that allows them to be compared. This points to a general vocabulary as the best solution. This experiment, however, illustrated that this is not an acceptable solution – if the argument terms are too general, the situation arises when the same argument can mean different things for different alternatives. If arguments are too specific, they cannot be compared.

One way to address these issues is to develop an extensible argument ontology where arguments can be supplied at varying levels of detail and can be compared at different levels.

The first question is what should be in the ontology. There needs to be a “default” ontology to serve as a baseline. The developer could extend this ontology to meet specific system needs. We also need to decide if requirements should be part of the ontology or if they should be treated separately. Another issue is how arguments should be classified. One possibility is based on type. Examples include: functional requirement, customer-specified non-functional requirement, other non-functional requirements, preferences (possibly designer-specific), and assumptions. Another possibility is the source: general, customer-specific, domain-specific. Temporal qualities may also be useful – is this an argument that will always apply or does it only apply under certain circumstances? Is there a way to measure the likelihood of it not holding in the future? Classifying arguments may be useful in supporting inferencing.

The next question is should the ontology be augmented with additional domain dependent information such as argument importance? While importance is likely to be different for each project, the ontology would be useful in propagating importance down the hierarchy to provide default values that can then be modified by the developer.

Another area of investigation is to determine if there is a way to assist the user in generating additional ontology entries. One reason for supporting a project-specific ontology is to try to eliminate (or at least reduce) the need for explanations for the argument-alternative relationship. It would be helpful if the system could support the user in breaking down the detailed explanations into specific arguments.

6.3 DECISION ANALYSIS

There are two different ways that decisions can be analyzed for common characteristics. One is by development phase – what are the typical types of decisions made at each phase. The other is by types of maintenance performed. First we will examine the decisions by phase.

In the Requirements Phase, there were two types of decisions made:

- What the requirements are;
- User interface design decisions.

In the Analysis Phase, there were decisions both about the system and about the process. These include:

- Which requirements do or do not require use-cases;
- What the analysis classes should be, and their types;
- Sequencing of interactions between the analysis classes;

In the Design Phase, the following types of decisions were made:

- What the classes are;
- Assignment or responsibilities to the classes;
- Coupling between classes (visibility);
- Inter-object message content and format;
- Choice between custom and language provided classes and methods.

Finally, the following types of decisions were made during the Implementation Phase:

- Data structures;
- Algorithms;
- Persistent storage methods;
- Refinements to use-cases.

Some decisions were made in later stages that probably should have been made earlier. An example of this is refining the use-cases. This involved adding information that had not been considered earlier in the development process.

Categorizing decisions based on the type of maintenance being performed is much more difficult since in this preliminary study there was only one change made of each type. Any categorization along the maintenance type axis would require more data collection.

6.4 INFERENCING OPTIONS

There are a number of different uses for inferencing over the rationale. InfoRat (Burge and Brown, 2000) used the rationale to check to see if the decisions made were well supported. Several other possibilities were suggested by this study.

One possibility would be to inference over the rationale to support perfective maintenance. Rationale is valuable during perfective maintenance because it is the only place where the consideration (or lack of consideration) of quality attributes is documented. Since perfective maintenance involves improving the software, there are a number of ways that the rationale can assist:

- Indicate which decisions should be reconsidered if quality priorities change.
- Indicate which areas of the system involved particular qualities in decision making (and may have a greater impact on the overall ability of a system to have a particular quality).
- Indicate areas where a particular quality attribute was not considered (and possibly should have been).

Another interesting use of inferencing over rationale would be to look for areas of the rationale where alternatives were tried and rejected. The rationale could be used to see if there were other decisions made for similar reasons that should be re-investigated. Rationale could also be used to keep track of what decisions were made based on reasons that are temporal in nature – i.e. are likely to change in the future. If the conditions change, the rationale can be used to determine where the design should change.

6.5 ENSURING RATIONALE USE

There are two difficulties with rationale use. The first, is *how* to use the rationale. There are a number of potential uses, some of which are described in the previous section. The second difficulty is how to ensure that the maintainer *makes use* of the rationale. The maintainer needs to be aware of when rationale is available and be able to easily access it. An even better approach would be to find ways of automatically displaying the rationale when needed.

Automatic presentation of rationale involves a number of issues. One is how and when to present the rationale to the user. This has to be done in such a way that it is useful yet not intrusive enough to hamper the development effort. A second problem is how to determine *which* rationale should be presented. The rationale needed may not be at the same level as the artifact currently being modified. In this case, it would be necessary to determine if rationale attached to a higher-level artifact (i.e., from an earlier development phase) needs to be shown to the user. For example, a decision may be made at the design level. Later, the maintainer may modify the code that implements that decision. It is not useful to only display the implementation rationale associated with the code, as the rationale that the user should really be seeing is the rationale for the design decision behind the code.

7. Summary and Conclusions

While this study was a good first step, and provided an initial research agenda, additional studies need to be performed in order to answer a number of questions. One is to determine how rationale could be most useful during maintenance. In this study the rationale was not *used* as much as we would have hoped during the modifications. This is more likely due to the types of changes made, rather than any indication of the usefulness of the rationale. More experiments need to be made. One gap in particular was exposed: there needs to be an extension/enhancement that modifies or removes an existing requirement, as opposed to the one made in this study that merely added a requirement. This could be used to see if the rationale can help to assess the affect on the existing code. It would also be interesting to see how the *type* of requirement (NFR vs. domain specific requirement vs. customer specific requirement) modified/removed affects the rationale.

Another reason for additional studies is to explore additional maintenance types. As mentioned earlier, there have been a number of attempts to define maintenance types. These range from the three intentions of perfective, adaptive, and corrective (Lientz and Swanson, 1980) to the twelve maintenance types described by Chapin (2000). While some types of maintenance may be of greater interest than others, it would be worthwhile to look at adaptive maintenance, since that was missing from this study.

Earlier, we analyzed the types of decisions made at each development phase. It might be interesting to also analyze the types of decisions made during different types of maintenance. If this becomes a goal of this research, it will be necessary to perform multiple studies of each type of maintenance to generate more data. That would allow us to compare the types of decisions made.

Design rationale has many potential uses yet has failed to live up to its full potential. By continuing with the research agenda outlined in Section 6, along with the additional studies outlined above, we hope to determine how design rationale can be most useful during software maintenance.

Acknowledgements

We would like to thank George Heineman for his discussions about how design rationale could be used to support the software design process.

References

Bañares-Alcantara, R., King, M.P., and Ballinger, G.: 1995, "Egide: A Design Support System for Conceptual Chemical Process Design," *AI System Support for Conceptual Design: Proc. of the 1995 Lancaster International Workshop on Engineering Design*, Springer-Verlag, New York.

Boehm, B. and Bose, P.: 1994, "A Collaborative Spiral Software Process Model Based on Theory W", *Proc. 3rd International Conf. on the Software Process,* IEEE Computer Society Press, CA, pp. 59-68.

Bose, P.: 1995, "A Model for Decision Maintenance in the WinWin Collaboration Framework", *Proc. of the Conf. on Knowledge-based Software Engineering,* IEEE Computer Society Press, CA, pp. 105-113.

Brooks, F.P. Jr.: 1995, *The Mythical Man-Month*, Addison Wesley, MA.

Burge, J. and Brown, D.C.: 2000, "Inferencing Over Design Rationale", *Artificial Intelligence in Design '00*, J. Gero (ed.), Kluwer Academic Publishers, Netherlands, pp. 611-629.

Chapin, N.: 2000, "Software Maintenance Types—A Fresh View," *Proc. of the International Conf. On Software Maintenance*, IEEE Computer Society Press, CA, pp. 247-252.

Conklin, J. and Burgess-Yakemovic, K.: 1995, A Process-Oriented Approach to Design Rationale, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, (eds), Lawrence Erlbaum Associates, Mahwah, NJ, pp. 293-428.

Dellen, B., Kohler, K., and Maurer, F.:1996, "Integrating Software Process Models and Design Rationales", *Proc. of the Conf. on Knowledge-based Software Engineering*, IEEE Computer Society Press, pp. 84-93.

Fischer, G., Lemke, A., McCall, R. and Morch, A.: 1995, Making Argumentation Serve Design, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, (eds), Lawrence Erlbaum Associates, pp. 267-294.

Grudin, J.: 1995, "Evaluating Opportunities for Design Capture", in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll (eds), Lawrence Erlbaum Associates, NJ, pp. 453-470.

Jacobson, I., Booch, G., and Rumbaugh, J.: 1999, *The Unified Software Development Process*, Addison-Wesley, MA

Karsenty, L.: 1996, An Empirical Evaluation of Design Rationale Documents, in *Proceedings of the Conference on Human Factors in Computing Systems*, Vancouver, BC, April 13-18.

Klein, M.: 1997, An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture, *Concurrent Engineering Research and Applications*, March, 1997, pp. 37-46.

Lee, J.: 1997, Design Rationale Systems: Understanding the Issues, *IEEE Expert*, Vol. 12, No. 3, pp. 78-85.

Lee, J.: 1990, SIBYL: A qualitative design management system, in *Artificial Intelligence at MIT: Expanding Frontiers*, P.H. Winston and S. Shellard (eds), Cambridge MA: MIT Press, pp. 104-133.

Lientz, B. P., Swanson, E. B.:1980, *Software Maintenance Management*, Addison-Wesley, Reading, MA.

MacLean, R.M., MacLean, A., Young, R.M, Bellotti , V., and Moran, T.P.: 1995, "Questions, Options and Criteria: Elements of Design Space Analysis", in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll (eds), Lawrence Erlbaum Associates, NJ, pp. 201-251.

Myers, K., Zumel, N. and Garcia, P.: 1999, Automated Capture of Rationale for the Detailed Design Process, *Proc. of the Eleventh National Conference on Innovative Applications of Artificial Intelligence*, AAAI Press, Menlo Park, CA, pp. 876-883.

Peña-Mora, F. and Vadhavkar, S.: 1996, Augmenting design patterns with design rationale, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing,* 11, Cambridge University Press, pp. 93-108.