Ph.D. Dissertation Proposal

Software Engineering Using design RATionale
(SEURAT)

Janet E. Burge
(Ph.D. Candidate)

Department of Computer Science
Worcester Polytechnic Institute

January 20, 2002

Advisor:
Dr. David C. Brown – Department of Computer Science, WPI

Committee:
Dr. George Heineman – Department of Computer Science, WPI
Dr. Carolina Ruiz – Department of Computer Science, WPI
Dr. Feniosky Peña-Mora – Massachusetts Institute of Technology, Cambridge, MA
# Table of Contents

1. Motivation ................................................................................................................................. 4  
   1.1 Why is Design Rationale Valuable? .................................................................................. 5  
   1.2 Why isn’t Rationale Used Now? ..................................................................................... 6  
   1.3 The Challenge of Using DR in Software Engineering .................................................. 7  

2. Problem definition ..................................................................................................................... 9  
   2.1 Design Rationale and the Software Development Process ........................................... 9  
   2.2 Uses of Rationale in Software Design .......................................................................... 11  
   2.3 Encouraging DR Use in Software Design ..................................................................... 13  
   2.4 Summary ......................................................................................................................... 13  

3. Relevant research ..................................................................................................................... 14  
   3.1 Design .............................................................................................................................. 14  
   3.2 AI in Design ..................................................................................................................... 15  
   3.3 Design Rationale ............................................................................................................. 16  
      3.3.1 Design Rationale Representation .......................................................................... 16  
      3.3.2 Design Rationale Capture ..................................................................................... 17  
      3.3.3 Design Rationale Use ......................................................................................... 19  
      3.3.4 Design Rationale for Software Design ............................................................... 20  
   3.4 Software Development ..................................................................................................... 21  
      3.4.1 Software Development Processes ......................................................................... 21  
      3.4.2 Requirements Engineering .................................................................................... 22  
      3.4.3 Software Architecture ......................................................................................... 23  

4. Investigating DR Uses: Inferencing over Design Rationale .................................................. 25  
   4.1 Using Rationale for Validation and Evaluation ............................................................... 25  
   4.2 Prototype System for Inferencing Over Rationale .......................................................... 25  
      4.2.1 Approach .............................................................................................................. 25  
      4.2.2 Inferences ............................................................................................................. 28  
      4.2.3 Vocabulary .......................................................................................................... 31  
      4.2.4 Tradeoff Evaluation .............................................................................................. 32  
   4.3 Implementation and Examples ......................................................................................... 33  
      4.3.1 Browse Rationale .................................................................................................. 33  
      4.3.2 Modify Rationale .................................................................................................. 35  
   4.4 Summary ......................................................................................................................... 36  

5. Software Development Process and Rationale ....................................................................... 38  
   5.1 Study Goals ....................................................................................................................... 38  
   5.2 Study Description ............................................................................................................. 39  
      5.2.1 Initial Design ......................................................................................................... 39  
      5.2.2 Corrective Maintenance - Minor Bug in the Program ......................................... 40  
      5.2.3 Perfective Maintenance - Revisiting the Design for Usability ......................... 41  
      5.2.4 Enhancive Maintenance - Extending the Functionality ..................................... 41  
   5.3 Study Results .................................................................................................................... 41  
      5.3.1 Initial Design ......................................................................................................... 41  
      5.3.2 Corrective Maintenance ....................................................................................... 43  
      5.3.3 Perfective Maintenance ....................................................................................... 43  
      5.3.4 Enhancive Maintenance ..................................................................................... 44  
   5.4 Summary and Conclusions ............................................................................................... 44  

6. Proposed Approach ................................................................................................................. 45
1. Motivation

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 (DR) 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 [Sim & Duffy, 1994]. DR is invaluable as an aid for revising, maintaining, documenting, evaluating, and learning the design.

Most work on design rationale has concentrated on capture and representation. Capture refers to the recording of the design rationale, either during or after designing. There are a number of methods proposed for capture, ranging from capturing design discussions on video tape to requiring that the designers manually record each decision as it is made. The amount of data captured also varies — some systems take a “kitchen sink” approach and record everything that may be of interest while others are more focused.

Representation of design rationale has also been studied extensively. Design rationale representations range from formal to informal. A formal approach allows the computer to use the data but does not always output information in a form that a human can understand. In addition, it requires that data be provided to the system in a rigid format. An informal approach provides data in formats that are easily generated and understand by a human but can not easily be used by the computer (e.g., natural language). Semi-formal approaches attempt to use the advantages of both approaches.

The key to making the capture worthwhile, as well as providing requirements for DR representation, is the use for, and usefulness of, the rationale. Capturing large amounts of detailed rationale is not useful if it is never looked at again. If rationale is useful to a designer, there is a greater incentive for the designer to assist with the capture of the needed information, particularly if that designer can immediately use the rationale. Also, knowing how the information will be used provides guidance about what information should be captured and how it should be represented. Kartensky [1996] studied the use of design rationale documents and found that while DR was useful for some designers, it was not sufficient to answer all of the designers questions. It may be possible to increase the usefulness of the rationale if it can be collected for a specific use, rather than as general documentation.

There are many potential uses including:

- **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 that may affect design quality.
• **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.

Unfortunately, despite all these potential uses, there are very few concrete examples of actual use. Much of the current research is on ways to capture and represent the rationale. These are important areas, but their value depends on how the resulting rationale can be used. This work will begin by focusing on the uses of DR, and then address capture and representation as needed to support those uses.

In this proposal, we describe how we will investigate ways that DR can be useful during software development. Specifically, we will be studying uses during software maintenance. Software development is an interesting application for DR in a number of ways. In one sense, software development is really system development where the software is but one piece of the resulting delivered system. The larger the system, the more decisions will be made during its creation. Also, because of software’s mutability, design decisions are more likely to be changed during software development than in other types of product development. Many changes will also occur during the maintenance phase as problems are discovered and fixed, as the system is adjusted to the meet the changing needs of the user, and as it is adapted to respond to changes in the underlying technology.

### 1.1 Why is Design Rationale Valuable?

As suggested above, there are a number of reasons why design rationale can be valuable. If design rationale were available, designers revising a design could use it to determine the original designer’s intent, as well as determining what alternatives had already been considered and why they were rejected. This can help to avoid duplicating work that was done on a previous iteration through the design. In some cases, the original reasons for making a decision may no longer be valid, and choosing a different alternative may be preferable. For example, one part may have been chosen over another because there was a surplus of them and the price was low. If this is no longer true, another part might be more cost effective. Preferences may also change if there is a new technology or an improved...
process available that make some decisions more, or less, desirable in new design iterations. Rationale can also serve as a form of “corporate memory” providing valuable insight into a design that would otherwise be lost if designers leave the company [Peña-Mora and Vadavkar, 1996; Brice and Johns, 1998].

Having this “corporate memory” is especially valuable for 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). Software 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.

While software may not have “parts” in the same sense as a mechanical object, it may rely on re-usable components or applications that change (or fail to change) as the technology advances. Rationale can be used both to determine when an upgrade or substitution in one of these components should or should not be made and the impact of that change on the rest of the system. The reliance on components or outside applications increases as the need to produce more complicated software systems more quickly and cheaply continues to escalate. Using rationale to capture the reasons for the choice of components and exactly how they are being used in the system can ensure that well supported choices are made and can indicate when adjustment may be needed in the future.

1.2 Why isn’t Rationale Used Now?

If rationale has such potential value, then why is it not in widespread use? There are a number of reasons why there are few, if any, successful DR systems in existence. 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. Because it is time consuming and viewed as documentation, DR capture may be viewed as expendable if deadlines are an issue [Conklin and Burgess-Yakemovic, 1995].

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]. There may also be issues with liability if potential problems with a decision are recorded and shown later to be the cause of a catastrophic failure of the system. DR that documents the reasons for the problem could be used against the designer [Conklin and Burgess-Yakemovic, 1995]. An even more frightening possibility 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].
Another issue affecting the likelihood of designers recording their decisions is that those who take the effort to record the decisions are unlikely to be those benefiting from them. This provides very little incentive to take the extra time and effort to record the DR, especially given the potential drawbacks of liability, disruption in the design process, and schedule impact stated previously. In some cases, the personnel doing the maintenance on the system may even belong to a different company than personnel who did the initial development, which gives the original developers even less incentive to record the rationale [Grudin, 1995]. It is not uncommon for companies to compete for the maintenance contract, which can be worth a great deal of money over an extended period of time. Anything that makes it easier for a new company to understand the system may weaken the developing company’s case for why they are best suited to have the maintenance contract, since the developing company’s superior knowledge of the system is usually their main argument as to why they are most qualified for the maintenance contract.

There are also issues with developing a useful representation. The information needs to be represented in a form that supports easy access and interpretation. Choosing a representation is a tradeoff between an informal representation that takes less effort to record and is easily understood by a human reader and a formal structured representation that lends itself to manipulation by a computer program. While natural language rationale is readable, the readability is only useful if there is a way to ensure that it can be retrieved and examined. On the other hand, translating the designers’ thoughts into a structured format is both labor intensive and likely to result in lost information and interpretation errors.

1.3 The Challenge of Using DR in Software Engineering

The primary focus of this work is to determine when and how rationale can best be put to use. Rationale can be captured and used at many different stages in the software life-cycle. For the designer, the incentive to record the rationale is inversely proportional to the time between capture and use. If capturing rationale provides immediate payback then designers are more likely to record it. For the project manager, rationale becomes more valuable over time since the original designer is less likely to be available or remember why they made the decisions. In this case, the person using the rationale is less likely to be the one who recorded it. This contradiction can only be resolved by either making the rationale immediately useful or by having rationale capture and use be an integral part of the development process, where the designer knows that the rationale is essential to someone, even if not immediately useful to him/herself.

Another difficulty with software design rationale is that it is not clear what software design rationale is. For mechanical design, it is easy to envision choosing between alternatives such as different types of materials. For software, an analogy is choosing between different algorithms. The difficulty with this analogy is that except in extremely time- or space-critical systems, choosing between two different algorithms that ultimately perform the same function may not have the same impact on the resulting system as the choice of material in mechanical design. Many of the key software decisions do not have an “outward” impact on the final product but instead affect the development process itself.
These decisions include choice of language, type of persistent data storage and development platform. While these may not have an externally visible impact on the system, they have a huge impact on how much the software will cost, how long it will take, and how easy it is to work with in the future.

In order to implement a system that uses rationale to assist the software maintainer, we need to investigate issues such as:

- **What are the important design decisions?** Are there ways of categorizing them?
- **What types of decisions are most likely to require revision?** When in the development process are these decisions made? When are they re-examined?
- **What relationships are there between rationale for decisions at different development phases?** Do they differ in level of abstraction? Are there likely to be contradictions? Do changes at one level require propagation to earlier or later levels?
- **How can rationale use and capture be integrated into the design process?** Can capture be performed with a minimum of overhead? Can rationale use be encouraged so that the maintainer takes advantage of it?
- **How can rationale be used?** Is it useful enough to be worth the trouble? Is some rationale more useful than others?
- **How can rationale be maintained?** Rationale for maintenance decisions also needs to be captured. How does this rationale affect that captured earlier? Can rationale modifications be propagated?

In the following sections, we describe the problem of using design rationale in software maintenance in detail. We also look at related work in the fields of design, design rationale, and software engineering. We describe a prototype system developed to demonstrate how design rationale can be used to verify and evaluate a design. We also describe an initial case study examining rationale generated during development and maintenance. We then describe our approach to building a system to assist with the capture and use of rationale, give a plan for the work involved in completing this system, and describe the contributions of this work. We then conclude with a schedule and references.
2. Problem definition

As described earlier, while everyone agrees that design rationale is useful, it is still not used in practice. Reasons for this are that the uses proposed for rationale are not compelling enough to justify the effort involved in its capture and that there are no, or few, systems available to support rationale use and capture. Software development is one area where we feel that rationale use could be beneficial and should be used. In particular, we feel that rationale would be especially valuable during software maintenance. For this reason this dissertation will address the following research problem:

Can rationale improve the effectiveness and efficiency of software maintenance?

We will discuss three areas that are important in understanding this problem:

– design rationale and its relationship to the software development process (2.1),
– potential uses for design rationale in software development (2.2), and
– how design rationale use could be encouraged (2.3).

2.1 Design Rationale and the Software Development Process

The software development process has multiple phases. While there are a number of different well-established software development processes in common use, they generally have the same phases in common (although terminology may differ):

• Analysis: this phase is where the needs of the customer are examined and transformed into requirements for the resulting system.
• Design: this phase is where the requirements are used to define the structure and behavior of the software system.
• Implementation: this phase is where the system is implemented, based on the design.
• Testing: this phase is when the system is tested to ensure that it functions properly and meets all the requirements defined in the analysis phase.
• Maintenance: this phase is when the system is running in the field and changes are made periodically to correct problems and/or add new functionality.

These phases do not necessarily take place in a linear format. Some software development processes are performed in an iterative fashion where small portions of the final system pass through design, implementation, and testing either in parallel with each other or in sequence. Variations on the software development process are described in more detail in section 3.4.

Design rationale could be generated at any stage of the design process and 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.

Figure 1 shows the development phases and the rationale that could be generated during each of them. Capturing all this information would present a significant amount of overhead to the software developer.

![FIGURE 1. Software Development Phases and Rationale](image-url)
While only one phase is explicitly named design, design activity, and decisions requiring rationale, occurs at all phases of the software development process. In addition, the rationale for these decisions may be required at a later phase in the process or, in the case of an iterative process or re-use of an existing system, at earlier phases in a new system design or iteration.

One area that needs more investigation is the relationship between rationale collected at different development phases. For some cases, the rationale at a later phase will be an elaboration on the rationale collected earlier. For example, an object class defined in the analysis phase may be split into several in the design phase. In some cases, the decisions could be less dependent on the earlier decisions. For example, off-the-shelf components used in the system may not be chosen until implementation time (assuming this choice is compatible with the overall design).

What if a decision made at a later phase does affect an earlier decision? There may be more information available then there was earlier. In this case, the new decision affects both the design and the design rationale. One difficulty in maintaining a software design (or more accurately, the software design description) is that as changes are made at lower levels, the developers often fail to go back to the design documentation and update it. As decisions change, the rationale changes also will require this backwards propagation.

### 2.2 Uses of Rationale in Software Design

There are many ways that rationale could be used in software design. The following paragraphs describe some uses that will be considered during this work:

1. **Rationale as Documentation**: Having the reasons behind the decisions recorded can be invaluable as people leave and join the software team. This would allow the knowledge of those leaving to still be available to the newcomers. The software development cycle is often very long and turnover can be quite high. This makes the availability of rationale particularly valuable.

   Another way that rational can assist in documenting software is to use it to provide traceability between different design phases (traceability refers to the ability to trace a requirement, or other element of the design, forwards and backwards through the phases of development). This is an aspect of design documentation that is often missing. It is very important to ensure that all requirements are met by the system and that no requirements are invalidated by making changes to the software in the future. If design rationale captures the relationships between decisions made at different stages in the design then these problems can be averted.

2. **Rationale for Revision**: Rationale can also be very useful when a design is revised, either due to a change in requirements or due to an error in the initial design. By recording those alternatives considered but not selected, rationale provides two useful services to the designer: it indicates which alternative decisions are not good, and the reasons, and also provides a list of decisions that were not chosen but that may be worth
a second look. This is information that would have to be painfully recreated by trial and error if it were not present in the rationale. Presence of the rationale both helps avoid repeating bad decisions and overlooking potentially good ones.

Because rationale captures the relationships between decisions, it can also be used to analyze the impact of design changes (revisions). The rationale can be used to determine which upstream and downstream decisions would need to be revisited if the proposed design change were made. This impact analysis is very valuable both in giving insight into how difficult the change is likely to be and by ensuring that all the affected portions of the design are known so that they can be changed as needed.

3. **Rationale for Reuse:** Rationale is also valuable when a design is reused. Generally, this involves some modification of the reused design either due to changing requirements or changing technology. If rationale provides traceability, the design can be analyzed by determining which portions of it are a result of which requirements. This helps to provide the information needed to assess the impact of a requirement change and to determine where changes need to be made to the design.

Rationale can also assist in changes needed if the technology changes. The reasons given in the rationale can be used to infer where decisions were driven by the technology available at that time. This information can be used to see where the design requires modification to exploit new technology and indicate if decisions rejected previously should now be reconsidered.

4. **Rationale for Validation:** Validating the rationale involves making sure that the rationale is complete. The reason why this is valuable is that well documented decisions may be more likely to be well thought out decisions. If the rationale is complete then the decision making process is more likely to have been thorough than if rationale was only partially entered.

There are several ways in which rationale can be validated:

- Is there an alternative selected for each decision required or do some decisions still need to be resolved?
- Are there decisions recorded that have no alternatives?
- Can each requirement be traced to a selected alternative?
- Were any alternatives chosen where only reasons for rejecting them were recorded?
- Are there any alternatives that have no reasons for choosing or rejecting them recorded?

While missing rationale does not necessarily mean that decisions were poorly thought out, the presence of rationale is a good indication that some thought went into the decisions and that multiple alternatives were considered.

5. **Rationale for Evaluation:** Rationale can also be used to evaluate the quality of the design. This evaluation is done by using the strengths and weaknesses of the arguments in the rationale to determine whether the decisions made were well supported. If not, then this indicates that either the designer did not make the best decision or the designer did not fully and accurately record his or her reasons. This could be a simple omission, or it may indicate that there are reasons that the designer prefers to not admit to. One
example of this would be a software designer choosing a particular programming lan-
guage because he or she wants a reason to learn it to increase their marketability, even
though the lack of experience in the language is a valid argument against the choice.

2.3 Encouraging DR Use in Software Design

In order for DR to be useful, it needs to be used. This is not as simple as it sounds. This
problem has two main issues that must be addressed. The first is to ensure that the DR is
available and easily accessed by the designer. The designer needs to know when there is
DR available without having to go out of their way to hunt for it. Also, the designer needs
to have the tools available that make DR use a benefit, not just another time-consuming
task.

The second issue is how to encourage the use. It is not enough to assume that if the DR is
available and there are tools to access it then the designers will use it. There needs to be a
way of enforcing, or at least encouraging, the designer to use the DR. One way is to
integrate DR use into the development process so that when DR should be used it is. If the
DR support tools are integrated into tools already used by the designers then it might be
possible to present the DR when it is needed without extra effort from the designer. This
would be the most automated approach and is preferable to a manual one where DR use
must be indicated before a task can be considered complete. The automated approaches
range from simply presenting a designer with related rationale at the appropriate time or a
more active approach where the past rationale is used to evaluate current decisions and
provide feedback to the designer.

2.4 Summary

As described above, crucial decisions are made at many points in the software
development process. Documentation of these decisions, and the rationale for them, could
be very useful in both subsequent development phases and in developing new systems
with similarities to the current one. The obstacle to this is that the rationale is not easy to
capture, and while there are many potential uses, the tools and methods are not in place to
support them.

One area where rationale could be especially useful is during software maintenance. The
goal of this work is to study how rationale could be used during maintenance and to
provide tools and methods that use design rationale to support the software maintainer.
3. Relevant research

It would be nearly impossible to cover in depth all the relevant work in all the areas connected to the problem we intend to address. Therefore we will have to look primarily at the significant contributions in four related directions:

– Design
– AI in design
– Design rationale
– Software design.

3.1 Design

There are a number of interpretations of the meaning of design. Hubka and Eder [1996] state that design can be interpreted as a noun, indicating the structure of artifacts and systems, or as a verb, indicating a process. The latter definition is particularly appropriate to this work:

“a process of establishing which of several alternative ways (and with what tools) things could be done, which of these is most promising, and how to implement that choice, with continual reviews, additions and corrections to the work — designing” (p. ix)

This definition is not complete — it does not indicate where the alternatives come from. It also, while indicating that the “most promising” alternatives should be used, does not indicate what that means. Tong and Srim’s definition [1992] emphasizes the requirements placed on the design — the need to conform to a specification, meet certain criteria (such as performance criteria and resource constraints), and work within constraints such as technology and time limitations. The conformance with requirements and constraints is what will make an alternative “most promising.”

Despite being overly simplistic, Hubka and Eder’s definition is important for two reasons. First, it clearly states that design is about selecting from alternatives. Second, it indicates that design continues past the specification and into the implementation. This indicates that designing is not limited to a phase of development that takes place between defining the requirements and building the product, it is something that continues throughout the product life-cycle. Interpreting design to apply to the entire cycle is also done by Pugh [1991] who defines a design core that starts with marketing (identifying customer needs) and continues through selling the resulting product.

Design can be categorized along many dimensions. One is by the design domain. Different domains, such as civil engineering design, mechanical engineering design, architectural design, and software design, while sharing many common issues, also bring their own unique difficulties to the design process. Another is by its level of abstraction, or stage in the design process, such as conceptual design or detailed design [Pugh, 1991]. Another way is by the amount of creativity involved in the design process (creative vs. routine design) [Brown & Chandrasekaran, 1989]. Still another is by what task or tasks must be
accomplished in order to complete the design. Parametric design [Brown, 1992] involves assigning values to known parameters while configuration design [Brown, 1998] involves determining how known components can be connected together to achieve the desired result. These dimensions overlap considerably but help to indicate how, if, or to what degree the design task can be automated as well as providing insight into the types of knowledge required.

3.2 AI in Design

While automating design sounds like a good idea, it is not a simple task. Design is considered an “ill-structured problem” [Tong & Sriram, 1992] where the mapping from requirements to implementation is not straightforward. Because automating design is not easy, design researchers have turned to AI techniques. Over the past twenty years, work has been done in applying AI techniques to a variety of design tasks and a variety of design problems.

As with design itself, AI in design research can be categorized among a number of overlapping dimensions. Some systems target specific application domains. These include circuit design [Mitchell, et. al, 1985; Mostow, et. al., 1992], mechanical design [Murthy & Addanki, 1987; Goel, 1991; Joskowicz & Sacks; 1999; Navinchandra, et. al, 1991; Araya & Mittal, 1987; Dixon, et. al, 1986; Ramchandran, et. al., 1988], algorithm design [Kant, 1985], and even table design [Bentley & Wakefield, 1995]. Other systems, such as TEAM [Lander & Lesser, 1992], and DIDS [Runkel, et. al., 1992] are domain independent.

The type of design performed can be categorized (although subjectively) by the amount of creativity or innovation involved. Some systems perform design that is more routine: Brown & Chandrasekaran’s AIR-CYL [1989] falls into that category. Others strive for more creative or innovative designs: William’s [1992] interaction-based design approach, for example, aims at creating novel devices.


A variety of AI techniques are used in design, often in combination with each other. Case-Based reasoning is used by CADET [Sycara & Navinchandra, 1992] and KRITIK [Goel, 1991]. Constraint-based reasoning is used by PROMPT [Murthy & Addanki, 1987] and PRIDE [Mittal & Araya, 1992]. Planning is used by MOLGEN [Stefik, 1981]. Machine learning is used in A-Design [Cambell, et. al., 1998] and LEAP [Mitchell, et. al., 1985]. Many systems use rules in some form, as well as other general AI techniques such as search and backtracking methods.
3.3 Design Rationale

Design rationale also falls into the category of AI in Design because at its core it is a knowledge representation problem. There are many ways of categorizing design rationale techniques and systems. In the following subsections, we describe related work by examining design rationale representation, capture, and use. We also examine DR systems developed specifically for software design.

3.3.1 Design Rationale Representation

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 formal approach allows the computer to use the data but does not always output information in a form that a human can understand. In addition, it requires that data be given to it in a more rigid format. An informal approach provides data in forms that are easily generated and understand by a human but can not be used by the computer. A compromise is to store information in a semi-formal representation that provides some computation power but is still understandable by the human providing the information. Semi-formal representations are often used to represent argumentation.

Informal Representations

Representations are categorized as informal when they capture information in the form generated by the designer during design, rather than requiring a new structure to be used. An example of a system using an informal representation is the Process Technology Transfer Tool (PTTT) [Brown & Bansal, 1991]. This tool, developed to capture manufacturing process design information, falls into the informal representation category because it records information in the original form in which it is created/used by the designer. This information is in the form of design documents (reports, process sheets, experiment descriptions, etc.) that are generated during the design of a manufacturing process.

Formal Representations

Formal representations contain information in a machine-readable format that makes it easy to manipulate and interpret the information using a computer, but is less easily understood by a human. One example of a system using a formal representation is M-LAP [Brandish, et. al., 1996], a machine-learning apprentice system that is integrated with design tools. It records user actions at a low level and then forms them into useful sequences. These sequences are parts of tasks, which are then parts of higher level tasks, etc. These sequences are formed using machine-learning techniques. These sequences are classified as a formal representation because sequences of mouse-clicks and movements are not generally understandable to a human. Another example of a system that uses a formal representation is Gruber’s Device Modeling Environment (DME) [Gruber, 1990]. In this system, the reasons behind the design are stored as part of a model of the designed device and are only accessible through the DME system.
Semi-Formal Representations

As described earlier, semi-formal representations are typically in the form of argumentation. The origins of argumentation theory are described in Stumpf & McDonnell [1999]. One of the earlier argumentation notations was developed by Toulmin [Toulmin, 1958] [Shumm & Hammond, 1993]. In this notation, an argument consists of a datum, which is a fact or observation, a claim, an claim resulting from that argument, a warrant, an argument supporting the claim, a backing, additional information supporting the warrant, and a rebuttal, an argument specifying exceptions to the claim. This was the origin of many currently used argumentation notations.

A number of semi-formal notations form the basis of design rationale approaches and systems. Design Space Analysis (DSA) uses the Questions, Options, and Criteria (QOC) representation [MacLean, et al., 1995]. This notation is used by Desperado [Ball, et al., 1999]. QOC represents the argumentation as questions, options, and criteria for choosing the options.

Another notation is Issue Based Information Systems (IBIS), used by gIBIS (graphical IBIS), and itIBIS (text based IBIS) [Conklin and Burgess-Yakemovic, 1995]. IBIS represents the argumentation as issues, positions, and arguments. IBIS is the basis of another notation, PHI, that is used in JANUS [Fischer, et al., 1995]. PHI captures similar concepts to IBIS but links them together differently. A number of systems use IBIS. These include KBDS [Bañares-Alcantara, et. al., 1995], a design support system for chemical process design, and DRAMA [Brice & Johns, 1998], a design rationale tool used in process engineering.

There have also been many notations created for specific DR tools. Examples of this are DRCS (the Design Rationale Capture System) [Klein, 1992] and DRIM (Design Recommendation and Intent Model) [Peña-Mora, et al., 1995]. DRCS represents argumentation using entities and claims about the entities. DRIM is used in SHARED-DRIM, which captures recommendations, justification, and intent for each participant in the design process.

3.3.2 Design Rationale Capture

Design rationale capture is a very difficult problem. There are a number of different methods that have been proposed. These include reconstruction, automatic rationale generation, apprentice, rationale as a methodological by-product [Lee, 1997] and also the historian approach [Chen, 1990]. The following paragraphs describe some of the approaches that use these methods. Capture methods are not mutually exclusive and a tool or approach may fall into several categories. It is also possible for a tool to fall into a different category depending on how it is used.

Reconstruction

Reconstruction consists of retrospectively creating the rationale after the design has been
complete. One example of this is Hyper-Object Substrate (HOS). HOS [Shipman & McCall, 1996] is a hypermedia representation of DR combined with knowledge-based system features. HOS captures design rationale by capturing design communication informally and then converting portions of that communication into a more formal representation. gIBIS [Conklin & Burgess-Yakemovic, 1995] was used retrospectively at NCR, although it could be used during the design process as well.

**Automatic Generation**

In this approach, design rationale is generated automatically from an execution history [Lee, 1997]. One example of a tool that does this is M-LAP [Brandish, et. al., 1996]. M-LAP is integrated with design tools and captures rationale by recording user actions at a low level and then forming them into useful sequences. These sequences are parts of tasks, which are then parts of higher level tasks, etc. These sequences are formed using machine-learning techniques. Another system doing this is the Rationale Construction Framework (RCF) [Myers, et al., 1999]. RCF uses its theory of design metaphors to interpret actions recorded in a CAD tool and convert them into a history of the design process.

**Apprentice**

In the apprentice approach [Lee, 1997], the system watches the actions taken by the designer and asks questions when it does not understand an action. In these systems, the rationale is, to some extent, pre-generated— if the designer’s actions match the system’s prediction then the system-generated rationale is saved. An example of an apprentice system is Active Design Documents (ADD) [Garcia, et. al., 1993]. ADD is a design rationale system for routine, parametric design. The designer uses the system to assign parameters for the system. If the designer’s recommendation matches the system, the system records rationale already built into the knowledge base. If there is a conflict between the designer’s action and the systems, the designer is informed and allowed to either modify the criteria, change their action, or override the system's recommendation.

**Historian**

In this approach, a person or computer program keeps track of all actions during the design process. This method is similar to the apprentice approach, except the system does not make suggestions. It is also similar to automatic generation except that the rational is specifically recorded during the design process, not generated later. An example of this approach is the Design History Tool (DHT) [Chen, et. al., 1990]. DHT records the constraints and decisions that occur from the initial design specification to the detailed design. This system is intended to both document and playback the design and design process.

**Methodological-Byproduct**

The idea behind rationale as a methodological-byproduct is for design rationales to “naturally emerge” from the design process [Lee, 1997]. What this actually means is open to
interpretation; certainly design processes designed around tools that automatically capture rationale (as described earlier) could be considered to produce rationale as a methodological byproduct. Another way would be to use a design process that forces rationale capture. This is done by Ganeshan, et. al. [1994]. In their method, the design description is modified only by changes to and refinements of the design objectives, thus capturing the rationale as part of the design process.

3.3.3 Design Rationale Use

There are a number of different ways that design rationale is used. Some systems only support retrieval of the rationale; how it is used after being retrieved is up to the designer. An example of a retrieval system is JANUS [Fischer, et. al., 1995], a design environment for kitchen design. JANUS consists of two components: a JANUS-CONSTRUCTION, used in making the design, and JANUS-ARGUMENTATION, that contains rationale from previous design cases. If the user makes a design decision that violates a design rule, JANUS-ARGUMENTATION is used to explain the rule and give examples of previous designs that did not violate the rule.

Some systems support retrieval and also offer the ability to check the rationale and/or the design for consistency and/or completeness. One example of this is SYBYL [Lee, 1990]. Services provided by SYBYL include maintaining consistency of the knowledge base as well as evaluating the alternatives based on claims for or against them. The Knowledge-Based Design System (KBDS) [King & Bañares-Alcantara, 1997] uses keywords to check the consistency of ISB networks that contain the rationale. C-Re-CS [Klein, 1997] performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. An Intelligent Design Evolution Management System (AIDEMS) [Thompson & Lu, 1990] uses constraint-based reasoning to check for inconsistencies and propagate revisions to a product description. In this system, the rationale consists of the explanations for the inconsistencies.

Another useful feature offered by some retrieval systems is the ability to ask questions about the design and/or rationale. ADD [Garcia, et. al., 1993] allows the designer to ask for explanations of why various parameter values were assigned. The designer can ask both “why” and “why not” questions. Gruber’s Device Modeling Environment (DME) [Gruber, 1990] supports a set of queries (pre-enumerated) that can be used to obtain explanations of design decisions. The Engineering History Base System [Taura & Kubota, 1999] also uses constraints to provide teleological and causal explanations of the designers thought processes. The Integrated Design Information System (IDIS) [Chung & Goodwin, 1998] integrates several systems together: a design system (using AutoCAD), a viewpoint system (supporting multiple views of the design and encouraging exploration), an issue-based system (containing the rationale), and a rule-based system (used to check for design violations).

Another use of rationale is to support collaboration: using the rationale as a means of communicating between different stakeholders in the design. An example of this is SHARED-DRIM. SHARED-DRIM is a system built using the Design Recommendation
and Intent Model (DRIM) [Peña-Mora, et. al, 1995]. Its main goal is to capture design rationale for use in conflict mitigation. SHARED-DRIM records design decisions and the rationale (argumentation) behind them and shares the information among the participating designers. By capturing rationale for each decision, and rationale for when decisions are not accepted, the design modification and approval cycle is shortened.

Rationale use can be taken yet another step further and used to assist in designing. An example of this is Reconstructive Derivational Analogy (RDA) [Britt & Glaadowski, 1996]. RDA is part of a larger system, the Circuit Designer's Apprentice (CDA). When CDA is given the requirements for a new electrical circuit, it searches a database of already designed circuits to find the closest match. If there are no circuits that match, or match after minor adjustments, RDA is used to create a design plan from the existing circuit. This design plan is then 'replayed' with the new requirements to create the new circuit design. One thing to note, however, is that the plan is a history of the design, and does not supply the reasons for any decisions.

3.3.4 Design Rationale for Software Design

Design rationale research ranges from general notations, such as IBIS [Conklin and Burgess-Yakemovic, 1995] to tools designed for specific types of design in specific domains, such as ADD [Garcia, et. al., 1993], which is specifically for parametric design in the HVAC domain. There has also been work done in software design. Potts and Bruns [1988] created a model of generic elements in software design rationale that was then extended by Lee [1991] in creating his Decision Representation Language (DRL), the language used in SYBYL. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale [Peña-Mora & Vadhavkar, 1996]. This system is used to select design patterns based on the designers intent and other constraints.

Co-MoKit [Dellen, et. al., 1996] uses a software process model to obtain design decisions and causal dependencies between them. These causal dependencies then are considered to be the design rationale. The system looks at tasks, goals to be reached during the process, products, the products produced (such as specifications), methods, the way that the task is accomplished, and agents, the human or computer who works on the task. When the process model is designed, the information flow between the tasks is captured and used to deduce the dependencies between them. The rationale consists of the justification for choosing a particular method. This is inferred from the dependencies in the model. For example, choosing a particular method may mandate specific values for system variables. The rationale for those variable assignments is the validity of the method choice. There are several types of justifications for a decision [Maurer, 1996]: justification based on validity or invalidity of previous decisions and justification based on validity or invalidity of existing parameter values.

CoMo-Kit, unlike the work proposed in this document, is not an argumentation system—it does not keep track of alternatives tried and rejected and the reasons behind them. Instead, it uses the dependencies to trace through the dependency chain to find the justification for
each decision. If a decision is changed, the system detects if this then invalidates the reasons for other dependent decisions.

WinWin [Boehm & Bose, 1994] is an approach aimed at coordinating decision making activities made by various “stakeholders” in the software development process. Bose [1995] defined an ontology for the decision rationale needed in order 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. The Bose ontology provides a relatively general argumentation structure for the decision rationale. WinWin focuses on the need for collaboration and primarily focuses on decisions made when determining requirements.

3.4 Software Design

Software design can be viewed in a number of ways. As with design in general (defined in section 3.1), software design encompasses the entire software life-cycle since design in some form is taking place at each point in the process. Design can refer to the design of the system, design of the code, design of the tests to be performed on the system, and so forth. Most software development processes have a phase called “design” as well, referring to the stage that takes place between determining the requirements and writing the code.

There are a number of research areas that relate to our work in Design Rationale. Three of them are discussed here:

• software development processes,
• software requirements analysis, and
• software architecture.

3.4.1 Software Development Processes

One way to encourage capture and use of rationale is to make it an integral part of the development process. Because software projects are often developed over a period of years, there has been a great deal of importance placed on using a well defined process for development. Process improvement initiatives, such as the Software Engineering Institute’s Capability Maturity Model [Paulk, et. al., 1993] have been underway in order to meet the primary software development goals of increased quality and reduced development costs [Osterweil, 1997].

Processes are commonly defined as software life-cycle models [Madhavji, 1991]. There are a number of different models that have been used over the past thirty years or so of software development. These include linear sequential models such as the step-wise model [Benington, 1956] and its more commonly known descendant the waterfall model [Royce, 1970]. In these models, the development process proceeds through a number of phases which may or may not have feedback loops between them.
Other models take a more iterative approach [Basili & Turner, 1975] which aim for incremental development. One iterative approach is the spiral model [Boehm, 1988]. This model is risk-driven—each time a development spiral is completed, the risk (likelihood of project failure) is examined. The number of tasks in the spiral varies: the variation proposed by Pressman [1997] shows a spiral starting with customer communication, then moving on to planning, risk analysis, engineering, construction and release, and finally customer evaluation. This is a high level model: within some of the phases, such as engineering, other processes could be used.

Besides the model development, other work has looked at process execution. Osterweil [1987][1997] suggested viewing software processes as software. He proposes implementing processes in coding languages to allow them to be expressed more clearly. Song & Osterweil [1994] developed a prototype system, Debus-Booch, in order to execute design processes based on the Booch Object-Oriented design methodology [Booch, 1991].

In order to encourage DR use in software development, the process chosen should be one that has been given wide acceptance in the software community. One process that has been widely supported is the Unified Process [Jacobson, et. al., 1999]. This process is an evolutionary process that grew out of a number of processes and modeling techniques. It is based on the object-oriented programming paradigm and is centered on use-cases [Jacobson, 1987]. Use-cases are descriptions of the dialog between the system and the user during an interaction. An interaction is when the user invokes the system, usually to perform a specific function, and terminates when the user has completed their action. The process is represented using the Unified Modeling Language (UML) [Rumbaugh, et. al., 1998]. One major advantage of this process is that it is has tool support — it is supported by Rational Corporation’s “Rational Rose” [Rationale, 1999].

### 3.4.2 Requirements Engineering

Requirements Engineering (RE) is a very active area of research. Zave [1997] defines it:

> “Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families.”

There are a number of aspects to RE where research is being done [Nuseibeh & Easterbrook, 2000]: eliciting requirements, modeling and analyzing requirements, communicating the requirements, reaching agreement on requirements, and evolving requirements. Each of these aspects has a variety of research taking place within it. Many of these either utilize rationale or could make use of rationale.

One common area of research is Requirements Traceability. Requirements traceability is the ability to follow, or trace, the life of a requirement forwards and backwards [Gotel & Finklestein, 1994]. According to Nuseibeh & Easterbrook [2000], requirements traceability can provide rationale for the requirements. This is certainly true of pre-requirement specification traceability [Gotel & Finklestein, 1994]: i.e. being able to trace to the origins of a requirement would give its rationale. Being able to trace back to the
requirements from the design would, in turn, give at least partial rationale for the design decisions. Work in requirements traceability is also valuable because many of the obstacles to requirements traceability are the same as those to rationale capture. These obstacles include the problems that requirements traceability is viewed as a low priority task, is not managed as a priority, has a cost/benefit imbalance, and more [Gotel & Finkelstein, 1994].

Another related area is Requirements Interaction Management [Robinson, et. al., 1999]. Requirements Interaction Management (RIM) is concerned with interrelationships between requirements. Often these relationships may involve conflicts and contradictions. It is necessary to be able to detect and correct these. There are a number of approaches to this problem. In one, ViewPoints [Easterbrook & Nuseibeh, 1995], the specification is broken into different views. These views may correspond to the perspectives of different participants in the development process. Working with different views can be very useful since the developer can focus only on the aspects of the system that he/she is concerned with. One danger, however, is that inconsistencies between views may arise. The ViewPoints approach detects these inconsistencies and allows development to continue without immediate resolution. It is assumed that at some point the inconsistencies will be resolved.

Another area related to RIM is work done on Non-Functional Requirements [Burge & Brown, 2002]. These requirements, also known as “ilities” or quality requirements, refer to overall qualities of the resulting system, as opposed to functional requirements which refer to specific functionality. The NFR Framework [Chung, et. al., 1995] represents the NFRs as goals that must be satisfied by the system. The design consists of a goal-graph giving the NFRs, alternative ways of satisfying them, and claims for and against these alternatives. This is very similar to other methods that use design rationale. One thing that is not always clear is how the NFR goal-graphs relate to the functional requirements.

One potential use of rationale is in evaluating trade-offs. Yen and Tiao [1997] represent requirements using fuzzy logic. These are referred to as “imprecise requirements” — this means that there is a range of values that satisfy them. When trade-offs are detected, the requirement values are adjusted to achieve the most satisfactory solution for the conflicting requirements.

Another Requirements Engineering area where rationale may be of use is in requirements scrubbing [Nuseibeh & Easterbrook, 2000]. In requirements scrubbing, requirements are removed in order to save cost. If the rationale for the requirements were available, it could be used to choose which requirements could most safely be eliminated. Additionally, if the mapping between NFRs and functional requirements was available (with rationale being one way to do this) then this could be used to both determine which requirements contributed toward cost and which contributed toward the various quality factors.

### 3.4.3 Software Architecture

Important decisions are made at all stages of the design process. Those made earlier are costlier to change than those made later [Pressman, 1997]. The importance of these
decisions implies that it is important that they are well justified. Capturing the rationale for these early decisions could help to ensure this. After determining the requirements, the next phase of development is designing the software architecture. The exact definition of what a software architecture is varies, but it is generally believed to be a high level of design giving an overall picture of the software system. Perry and Wolf [1992] present a model of software architecture with three components: elements (pieces of the architecture), form (relationships between the elements), and rationale (the reasons behind the architectural choices). Except for defining it as a necessary component, little has been done with rationale for software architecture.

One common approach to software architecture is the use of Architectural Styles [Garlan & Shaw, 1993]. There are a number of standard architectural styles in common use; one example is a pipe and filter architecture. Klein & Kazman [1999] look at architectural styles as ways of fulfilling certain quality attributes, or non-functional requirements. Each style is intended to satisfy specific quality attributes. Styles can be combined in order to obtain an architecture that best satisfies the quality attributes that are required. Use of architectural styles has also been proposed as a means of facilitating reuse [Medvidovic & Taylor, 1997].

Just as a number of architectural styles have been created to meet a variety of needs, different Architecture Description Languages (ADL) have been developed to represent them [Medvidovic & Taylor, 1997]. Some were developed to support a specific architecture. For example, the C2 architecture [Taylor, et. al., 1995] has its own ADL. An ADL can also be used for a specific type of system. Darwin [Magee, et. al, 1995], for example, is a notation for describing architectures of distributed systems. As there is unlikely to be any agreement on a common ADL, an architecture interchange language, ACME [Garlan, et. al., 1997] has been developed to translate between ADLs. It is also possible to represent an architecture in UML, either directly [Hofmeister, et. al., 1999] or by extending UML to mimic an existing ADL [Robbins, et. al., 1998]. It may be possible to augment ADLs in order to capture the rationale along with the architecture description.
4. Investigating DR Uses: Inferencing over Design Rationale

As discussed earlier, there are a variety of different uses for design rationale. The following sections describe work done on a prototype system that investigated using rationale for validation (of the rationale) and evaluation (of the design).

4.1 Using Rationale for Validation and Evaluation

Two interesting uses of DR are using it for validation of the rationale and evaluation of the design. Validating the rationale involves verifying that the rationale is structurally complete and that there are no obvious discrepancies, such as decisions made that had no arguments in their favor. Validation is important because completed rationale may indicate that decisions were well thought out – designers were able to explicitly justify their design decisions.

Rationale can be used to evaluate the design by checking to see if the decisions made were well supported. For example, if a decision has more, or stronger, arguments against it than for it then it may not be the best choice. Also, there may be alternatives that have more support than the ones that were chosen – this may indicate that there is either missing rationale or that the choice made should be reconsidered.

4.2 Prototype System for Inferencing Over Rationale

A prototype system, InfoRat (Inferencing Over Rationale) [Burge & Brown, 2000], was built to investigate how inferencing over rationale could support validation and evaluation using rationale. The following sections describe the approach and implementation.

4.2.1 Approach

In the InfoRat approach, design rationale is viewed as a bridge between design phases where the rationale can be used to trace through the decisions, starting with the requirements. The design begins with a set of requirements defining the system being designed. These requirements are then mapped to goals and, if required, sub-goals. Goals and sub-goals then can be satisfied by one or more alternatives. Each alternative then maps to an artifact, or a requirement for the next stage of design. The rationale for each choice is represented as arguments, expressed as claims, for or against each alternative. Figure 2 shows how design rationale links the requirements and the design.
The resulting rationale serves both to document the design and to provide a means for design verification. This verification involves ensuring that the design is consistent and complete, i.e., all requirements correspond to goals and all goals have selected alternatives. The following subsections describe the important aspects of this approach.

### 4.2.1.1 Example Problem

For illustrative purposes, a simple example of a traffic light design [Gogolla, 1998] was used. This was done to provide rationale that was simple to construct but rich enough to demonstrate the concepts. For more detailed information on traffic signal phase and cycle selection, see Zozayza-Gorostiza and Hendrickson [1987].

The traffic light example describes the conceptual design of the traffic lights for an intersection between two streets where one street had a heavier flow of traffic than the other, except during rush hour. This intersection also had frequent traffic turning from traveling South to traveling East. In addition to supporting those aspects of the intersection, the light system also had to be designed so that it would handle failure as safely as possible. Figure 3 shows the intersection.

![Intersection Diagram](image)

**FIGURE 3. Intersection Diagram**

This results in the following requirements for the traffic light system:
• Use four traffic lights
• Provide safe traffic flow
• Allow for heavier traffic on the North-South road
• Allow for traffic turning South to East
• Safely handle light failures

Each of these requirements can be satisfied in a number of ways. For example, choosing four traffic lights involves deciding what types of phases the lights should have, deciding if all four lights should be identical, and deciding if the lights should have arrows for turning or not. Providing safe traffic flow requires controllers for the lights to ensure that traffic can not be flowing on the E-W road at the same time that it is flowing on the N-S road. There are also a number of ways that the heavier traffic flow on the N-S road can be handled. Sensors can be used to monitor the flow of traffic or the lights can go to flashing yellow or red at times when traffic on the E-W road is lighter. Assistance for turning can be provided by delaying the lights or by using turn signals. There are also different ways that light failures that can be handled. One way is to shut down the intersection completely, although it might be better to turn it into a “four way stop” so that some traffic flow can still occur.

4.2.1.2 Representation

As described above, there are a variety of methods for representing rationale. In order to support inferencing, a structured or semi-structured representation is required. DRL [Lee, 1990] has the richest rationale representation of the systems studied. A meaningful subset of DRL was chosen to allow exploration of possible inferences and to keep the representation relatively simple. For DRL, the elements represented are artifact, requirement, goal, alternative, claim, group, viewpoint, and question. DRL also supports several relations between these elements including: is-a-part-of, is-a-subclass-of, is-argument-for, and is-argument-against.

The InfoRat system implements a subset of these elements: requirement, goal, alternative, and claim. It also allows several relationships: supported-by, sub-goal, alternative-for, argument-for, and argument-against. Figure 4 shows the elements represented in InfoRat and the relationships between them.
As the figure indicates, each goal can have multiple sub-goals, an alternative can be used to satisfy more than one goal, and a claim can be an argument for or against multiple alternatives. Figure 5 shows the goals as well as a partial set of alternatives and claims for the requirement to use four traffic lights.

When a claim is used as an argument for or against an alternative, it is given to values to indicate its influence on the design decision: an amount, between one and ten, that indicates “how much” the alternative meets the claim (i.e. how safe) and an importance, between zero and one (not important to essential), indicating how important the claim is when trying to meet the goal. This weighting scheme was used because it is simple yet is rich enough to support the types of inferencing desired for the system.

The amount and importance are multiplied together and then added to the ratings of other arguments to indicate the overall rating for an alternative. For example, if the alternative “Arrows” (as shown in Figure 5) has a claim in its favor of “Safety”, with an amount of seven and an importance of one, and claims against it of “NOT Affordability”, with a amount of five and an importance of .5 and “NOT Simplicity”, with an amount of four and an importance of .3, its overall rating would be 3.3:

For $\begin{align*}
7 * 1 \\
\end{align*}$ Against $\begin{align*}
5 * .5 + 4 * .3 \\
\end{align*}$ Result $\begin{align*}
3.3 \\
\end{align*}$

4.2.2 Inferences

The InfoRat system inferences over the rationale to check for completeness and consistency. The inferencing can be broken into two categories: syntactic inferencing that
uses the structure of the rationale, and semantic inferencing that looks at the contents/values of the different rationale elements.

Syntactic inferencing looks for the following inconsistencies in the rationale: requirements with no corresponding goals, and goals (or sub-goals) with no selected alternatives. The syntactic checks are primarily concerned with ensuring that the rationale is complete. Figure 6 shows the requirement “Four Traffic Lights” and its relationships. In this example, the goal “Select Type of Directionals” has two alternatives but neither has been selected. Figure 7 shows a syntactic check that looks to see if there are any requirements that do not trace to goals with selected alternatives. This check detects that the requirement “Four Traffic Lights” was not satisfied. Both these figures, as well as those that follow, show actual output from InfoRat.

FIGURE 6. Goals and Sub-goals for the Unsatisfied Requirement

<table>
<thead>
<tr>
<th>Requirement: Four Traffic Lights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goals:</td>
</tr>
<tr>
<td>Goal: Select Four Lights</td>
</tr>
<tr>
<td>Subgoals:</td>
</tr>
<tr>
<td>Goal: Select Types of Phases</td>
</tr>
<tr>
<td>Alternatives:</td>
</tr>
<tr>
<td>German 4-Phase Lights</td>
</tr>
<tr>
<td>Italian 3-Phase Lights (Selected)</td>
</tr>
<tr>
<td>Goal: Select Type of Directionals</td>
</tr>
<tr>
<td>Alternatives:</td>
</tr>
<tr>
<td>Light w/o Turn Signals</td>
</tr>
<tr>
<td>Light with Turn Signal</td>
</tr>
<tr>
<td>Goal: Select Light Configuration</td>
</tr>
<tr>
<td>Alternatives:</td>
</tr>
<tr>
<td>Mixed Light Types (Selected)</td>
</tr>
<tr>
<td>All Lights the Same</td>
</tr>
</tbody>
</table>

**FIGURE 6. Goals and Sub-goals for the Unsatisfied Requirement**

*****************************************************************
*          Verify Design Rationale            *
*****************************************************************

Choose one of the following:

1: Show Full Verification Report
2: Check for Unsatisfied Requirements
3: Check for Unsubstantiated Alternatives
4: Check for Non-Optimal Alternatives
5: Check for Contradictory Arguments
6: Check for Invalid Tradeoffs
7: Check for Consistant Arguments
8: Check for Incomplete Rationale

E: Exit Menu

Enter Selection: 2

 Unsatisfied Requirements:

None!
Semantic inferencing looks at the reasons for and against the alternatives. There are three types of discrepancies looked for at the argument level: selected alternatives where the arguments against the alternative outweigh the arguments for the alternative, (as shown in Figure 8), selected alternatives where the alternative selected is not the best choice, (as shown in Figure 9), selected alternatives where the same argument is used both for and against the alternative, (as shown in Figure 10).

**FIGURE 8. Arguments Against Outweigh For**

For Goal: Priority to NS Traffic
Selected Alternative: Configuration Changes w/Time
(Rating = -1.5)

**FIGURE 9. Best Alternative Not Chosen**

Selected Alternative: Mixed Light Types
(Rating = 1.5)
Best Rated Alternative: All Lights the Same
(Rating = 2.5)

**FIGURE 10. Contradictory Arguments**

There are two types of consistency checks made at the goal level: completeness, where alternatives are examined to ensure that the same arguments are considered for each alternative, as shown in Figure 11, and consistency, where alternatives for a specific goal are examined to ensure that a particular argument is given the same importance for each alternative, as shown in Figure 12.
4.2.3 Vocabulary

In order to support semantic inferencing, it is necessary to have a known vocabulary for claims (arguments for or against an alternative). The vocabulary consists of two categories: a pre-defined, standard vocabulary, and a user-defined, domain-oriented vocabulary. We refer to these as the Standard Claim Vocabulary and the User-Defined Claim Vocabulary respectively.

The Standard Claim Vocabulary is pre-defined to match the design task. For software design, a vocabulary has been built based on the “ilities” [Filman, 1998]. Figure 13 shows the Standard Claim Vocabulary used by InfoRat.

![FIGURE 13. Standard Claim Vocabulary](image)

Claims can be added to the User-Defined Claim Vocabulary at any time during the design process. These are arguments that are specific to the design project. Figure 14 shows the User-Defined Claim Vocabulary for the traffic light design problem.
4.2.4 Tradeoff Evaluation

Another type of semantic inferencing makes use of background knowledge to evaluate trade-offs. The background knowledge specifies two types of information:

- Does a causal relationship exist between two claims and;
- Which claim, if either, is more important.

If two claims are causally related, i.e. more of one means less of another, then InfoRat will check to ensure that these claims never appear on the same side of an argument. Figure 15 shows an example of a causality error. InfoRat also checks to ensure that if two claims are causally related they never appear individually. Figure 16 shows an example of a missing claim. In addition, if the background knowledge indicates that one claim is more important than another, this will also be checked for. Figure 17 shows an importance violation along with a causality violation.

---

**FIGURE 14. User Defined Claim Vocabulary**

**User Defined Arguments:**
- Starves one direction
- Optimizes Traffic Flow

**FIGURE 15. Causality Violation**

Tradeoffs are inconsistent for Goal: Select Type of Directionals
For Alternative: [Light w/o Turn Signals] inconsistent tradeoffs are:
- Causally related arguments Safety and Affordability appear on the same side of an argument and Affordability is rated as more important

**FIGURE 16. Missing Claim**

Tradeoffs are inconsistent for Goal: Select Types of Phases
For Alternative: [German 4-Phase Lights] inconsistent tradeoffs are:
- Argument Safety appears without normally opposing argument Affordability
Tradeoffs are inconsistent for Goal: Priority to NS Traffic
For Alternative: [Configuration Changes w/Time inconsistent tradeoffs are:

Causally related arguments Safety and Affordability appear on the same side of an argument and Affordability is rated as more important

FIGURE 17. Tradeoff Importance Violation

4.3 Implementation and Examples

InfoRat has been implemented in CLIPS [CLIPS, 1998] and performs three main functions: Rationale Browsing, Rationale Modification, and Rationale Verification.

4.3.1 Browse Rationale

The browse function is used to examine the rationale stored in the system. The designer can examine the status of each element and its relationship with other elements.

The first option, List DR Element Types, allows the user to quickly view the different DR elements currently in the system. Figures 18 through 20 show the element listings for requirements, goals, and alternatives.

Requirements:

- Four Traffic Lights (Satisfied)
- Safe traffic flow (Satisfied)
- Traffic heavier N-S (Satisfied)
- Frequent South to East Turning Traffic (Satisfied)
- Safely Handle Light Failures (Satisfied)

Goals:

- Select Types of Phases (Satisfied)
- Select Type of Directionals (Satisfied)
- Select Light Configuration (Satisfied)
- If EW traffic, no NS traffic (Satisfied)
- Safe Flow of Traffic (Satisfied)
- Priority to NS Traffic (Satisfied)
- Turn Assistance to SE Traffic (Satisfied)
- Select Four Lights (Satisfied)
- Stop all if Light Fails (Satisfied)

FIGURE 18. Requirement Listing

FIGURE 19. Goal Listing
The remaining options give the user a more detailed view of each element. Figure 5 (in Section 4) showed the information displayed about a requirement and its goals. Figure 21 shows the contents of an alternative, Blinking Red/Yellow.

Each rationale element contains a version number and a description of the element. The version number is used to keep track of changes in the rationale so that it can be determined if the state of any rationale element was changed during the design process. The description is used to describe the element to the user. InfoRat also allows the user to view the version history to see the changes made to the rationale and the reasons for the changes in the rationale. Figure 22 shows an example of a version history.

<table>
<thead>
<tr>
<th>Name: <code>&lt;Instance-always_blinking&gt;</code></th>
<th>Alternative: Blinking Red/Yellow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative for:</td>
<td>Priority to NS Traffic (Not Selected)</td>
</tr>
<tr>
<td>Claims For:</td>
<td></td>
</tr>
<tr>
<td>Claim: Simplicity</td>
<td>Applicability: IS</td>
</tr>
<tr>
<td>Amount: 3</td>
<td>Importance: MODERATE</td>
</tr>
<tr>
<td>Claim: Affordability</td>
<td>Applicability: IS</td>
</tr>
<tr>
<td>Amount: 4</td>
<td>Importance: MODERATE</td>
</tr>
<tr>
<td>Claims Against:</td>
<td></td>
</tr>
<tr>
<td>Claim: Safety</td>
<td>Applicability: NOT</td>
</tr>
<tr>
<td>Amount: 7</td>
<td>Importance: MODERATE</td>
</tr>
</tbody>
</table>

**FIGURE 21. Alternative Blinking Red/Yellow**
The first two changes were made in response to errors detected by InfoRat. The remaining three changes could either be responses to errors or inconsistencies shown by the system or in response to changes in the requirements. Notice that the reasons given for the first two changes are reasons for changes to the rationale, not reasons for changes to the design, i.e. design rationale rationale, not just design rationale.

4.3.2 Modify Rationale

InfoRat allows the user to modify the different DR elements. Figure 23 shows the modification choices.
For requirements, the user is allowed to add a requirement, delete a requirement, or change which goals are associated with the requirement. Goals can either be associated or disassociated with the requirement. If a requirement is deleted, the delete cascades, i.e., any goals, sub-goals, and alternatives that only relate to this requirement are also removed.

For goals, the user can add a new goal or modify a goal already in the system. Allowable modifications for existing goals are adding a sub-goal, deleting a sub-goal, adding an alternative, removing an alternative, or selecting an alternative. When an alternative is selected, any alternative for that goal that may have been selected earlier is deselected to ensure that only one alternative can be selected for a goal.

For alternatives, the user again has the option of adding a new alternative or modifying an existing one. For an existing one, the user must first specify which goal the alternative is for. This is required because an alternative can apply to more than one goal. The user is then presented with several options for changing the arguments for and against the alternative. Figure 24 shows the options for modifying alternatives.

For arguments, the only option is adding additional arguments. When each modification is made, the user is prompted for a reason for the change. This provides additional information that can be retrieved by the user as part of the version history.

---

**FIGURE 24. Modify Alternative Options**

| Target Goal: [Select Light Configuration] |
| Target Alternative: [Mixed Light Types] |

Choose one of the following:

1: Select the Alternative
2: Add an Argument for the Alternative
3: Add an Argument against the Alternative
4: Remove an Argument for/against the Alternative
5: Change the weight of an Argument for/against the Alternative
6: Change the importance of an Argument for/against the Alternative

E: Exit Menu

---

### 4.4 Summary

InfoRat supports a designer by inferencing over DR to check for completeness and consistency, as well as other problem indicators. This augments existing approaches, such as constraint satisfaction, that only reason about the design. Our work complements the work by Klein and by Lee on reasoning over design rationale.

A predefined vocabulary is provided so that the contents of the arguments can be used for inferencing. The user can extend this vocabulary by adding additional arguments that are more design problem specific. When the user modifies the design rationale, the system
prompts them for modification rationale. This combination of a standard, machine-interpretable vocabulary and user-supplied rationale allows the design history to be kept, and enables the system to reason over the rationale.

The concepts developed in this work, as demonstrated by the InfoRat system, provide a new and different way of looking at DR use. Intelligent reasoning over DR will provide more beneficial use for the collected DR than just its retrieval and presentation. Such reasoning can provide strategic guidance for the design process. In addition it can provide a novel way of checking for design quality, as designs with poor rationale are less likely to be of high quality. We believe that this research provides a new view of how to use Design Rationale whose development has great potential.
5. Software Development Process and Rationale

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. The following sections describe this study.

5.1 Study Goals

There were two different goals for this study. The primary goal was to determine a research agenda by studying how rationale was used and modified during several different maintenance tasks. A secondary goal was to gain a better understanding of what the rationale for the various software development phases looks like.

There are a number of different classifications for types of software maintenance [Chapin, 2000]. Three types were examined in this study: corrective, perfective, and enhancive. We used a simple meeting scheduler system as the software being maintained.

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.
5.2 Study Description

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 was written in Java and used a previously developed component (provided as an input to the Meeting Scheduler developer) as part of its user interface that allowed the user to enter meeting information into a schedule. 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 following sections describe the artifacts and rationale created for the initial design and each of the proposed modifications.

5.2.1 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 25 shows the graphical convention used in documenting the rationale.

This contained the following components:
During each phase of the development process, the applicable Unified Process artifacts were created along with the rationale for the decisions made during their creation:

- **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. For the Meeting Scheduler system, the rationale was recorded for the requirements developed and for requirements that were considered but rejected. 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 (more detailed) design decisions made while writing the code. This included detailed information about data structures and algorithms.

### 5.2.2 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 implementation level, the coding and use of the method. The rationale was updated to capture both the original decision and the alternative used to replace it.
5.2.3 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.

5.2.4 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 the user could move between by selecting different conference rooms.

5.3 Study 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.

5.3.1 Initial Design

Rationale was generated for each phase of the development process. Some observations, described in the following sections, were specific to design phases while others applied to the rationale overall.

5.3.1.1 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. An argument could 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 by capturing a large number of language-specific event traffic that, while necessary to the implementation, was not crucial to the design.

In the Implementation Phase, 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.

5.3.1.2 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 the 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 create 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.

5.3.2 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 observation about rationale was that it 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?

5.3.3 Perfective Maintenance

In this maintenance example, poor choices were made in the original user interface design that required some modifications to improve efficiency. 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 that 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 the user interface change was required. 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.

5.3.4 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 when adding the new requirements 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.

5.4 Summary and Conclusions

This study was a good initial start at fulfilling the two goals described in Section 5.1: establishing a research agenda and gaining a better understanding of what software design rationale looks like. It did not provide answers to all our questions but did suggest some areas where further study is needed. Section 6 describes the research agenda resulting from the study.
6. Proposed Approach

Sections 4 and 5 describe earlier work we did to investigate how design rationale could be used. The InfoRat work demonstrated some potential uses while the software maintenance study indicated some areas where our research should concentrate. This section describes the research agenda for the dissertation.

Our research will require examining the following questions:

1. *How can rationale be used to assist in software maintenance?* We hope to use rationale for retrieval, evaluation, traceability to requirements, impact assessment, and as a bridge to other affected portions of the design.

2. *How can decisions be captured with enough specificity to be useful yet still general enough to allow for inferencing?* We plan to develop a hierarchy of reasons for modification that can be used at different levels of abstraction to allow comparisons during inferencing.

3. *Does rationale differ for different types of software modifications?* We believe that rationale will be used and created at different levels of the design process depending on the type of modification.

4. *Does maintenance rationale differ from original rationale?* We expect maintenance rationale to be similar in structure, with reasons for change that are unique to modification.

5. *Are there portions of the design or phases of the design process (Figure 1) where rationale capture would be more useful than others?* We expect this to be true. Rationale at early phases is important because the decisions affect more of the resulting implementation, however these decisions are less likely to be altered than those later in the process.

6. *What is the relationship between rationale collected at different phases? Is it only via the design artifacts?* We have yet to see a direct relationship between rationale at different phases except via use of common arguments for and against decisions.

7. *How can rationale changes be propagated?* Changes in the rationale need to be propagated to indicate where they affect both the system being developed and the rationale previously stored for other design phases. While automated propagation would impose the smallest burden on the developer, automatic changes may not be the safest approach. We expect to develop a way to inform the maintainer of, and assist the maintainer with, the changes needed to keep the rationale consistent.

The following sections describe the major aspects of our research agenda and the system that will be used to illustrate the rationale representation and uses described in this proposal.
6.1 Additional Software Development Studies

The study described in section 5 was a good first step towards studying design rationale and its role in the software maintenance process. There are a number of areas that require further investigation. In particular, there needs to be additional investigation into the types of decisions made during the development process and which are more likely to require rationale. There are also additional types of maintenance that need to be examined in order to make the study more complete.

6.1.1 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 of the development process, 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 of development 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. Such iteration, however, is normal in software development.

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.

6.1.2 Additional Studies

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 our initial software maintenance study, the rationale was not used as much as we would have hoped during the modifications. This is probably due to the types of changes made, rather than any indication of the usefulness of the rationale. More experiments need to be conducted.

In particular, 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.

6.2 Representing the Rationale

A representation for the design rationale needs to be created to support the software maintenance process. This involves representing arguments in a way to support useful inferences. The study described in Section 5 provided much insight into what was required for a useful representation. The following sections describe some aspects of the representation that will require further work in the course of completing the dissertation.
6.2.1 Argumentation Structure

The simplified representation used in the study described in Section 5 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 as 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.

6.2.2 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 study, 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 whether the ontology should 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 Using the Rationale

As described earlier, the major focus of this work is on determining how rationale can be used. There are a number of aspects to this. One is to determine uses for the rationale. The other is ensuring that the rationale, when present, is actually used.

6.3.1 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.

6.3.2 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. This is not the major focus of our work but is important in developing a system that is useful to the maintainer.

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.

6.4 SEURAT Software Maintenance Support System

To drive and evaluate this research, we will develop a system that supports the maintainer. This system, SEURAT (Software Engineering Using RATionale), 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.

This system will involve several components. First, there are the software artifacts themselves. The system will be developed to support a modified Unified Process similar to the one described in Section 5. This will involve four stages of development: requirements, analysis, design, and implementation. During requirements, the developer will perform requirements analysis (i.e., writing the requirements) and initial user interface design. During analysis, the developer will create use-cases and collaboration diagrams. During design, the developer will create class diagrams. The sequence diagrams
described in section 5 will not be supported unless the additional studies described in section 6.1.2 indicate that they give some value beyond what is provided by the collaboration diagrams. During implementation, the developer writes the source code for the program, based on the results of the design phase. Each artifact produced during each phase will have rationale associated with it.

The rationale will be used in two different ways. One is as documentation. When the developer modifies a development artifact, the applicable rationale will be presented to them by the SEURAT Argument Editor. This is primarily a retrieval problem but is somewhat complicated because the useful rationale may not be associated with the specific artifact being modified but instead may have been captured at an earlier development stage. For example, a piece of code being modified is the implementation of the design created in the design phase. The decisions that most affect that code, and their rationale, may have occurred during the design phase.

The second use involves inferencing over the rationale to validate the rationale, evaluate the design, maintain consistency within the rationale, and ensure that errors are not repeated. This requires some form of knowledge-based system to perform this analysis. Rationale will be stored in a centralized repository that can be accessed by all components of SEURAT. This repository will maintain a history of changes to the rationale.

Figure 26 shows the proposed architecture for SEURAT. The development artifacts will be maintained by one or more CASE tools and/or text editors. The rationale will be entered using a special Argument Editor. The rationale will be stored in a rationale repository that is then used by the inference engine to perform analysis. Results of the analysis will be presented by the Argument Editor, which will also allow the user to make changes to the rationale to correct any defects found during analysis.
6.5 Discussion of the approach

There are a number of difficulties involved with determining the appropriate representation for the design rationale. The representation needs to contain sufficient detail to be useful but not so much that it can not be used in inferencing. The ontology of arguments proposed here is one step towards solving that problem. Another issue is the burden placed on the developer during collection. A natural tendency of the developer is to minimize the amount of time and effort spent on something that does not directly contribute toward a working system. This is another argument towards keeping the amount of information represented to the minimum amount of detail that is required in order to be useful.

Even a perfect representation (which is unlikely to exist) is not useful if the rationale is never used by the developer. The proposed system will provide the user with tools that perform useful inferencing over the rationale to look for problems. In addition, it will provide the rationale as documentation when the related software artifacts are modified. Again, there is a balance that must be struck between providing potentially useful information and harassing the user by continually making them look at the rationale.

One of the goals of the study in Section 5 was to see how rationale was used during software maintenance. This goal was not met: none of the three modifications made required using the rationale. The additional studies described in this proposal will provide
new insight into how and when software rationale is used during maintenance. The more information obtained about when rationale is useful, the more the resulting system can be targeted to collect information that is most likely to be useful and to present information when it is most likely to be needed.

The proposed system, SEURAT, will be used as a test-bed to demonstrate the effectiveness of the representation and uses proposed in this document. The system will be aimed at supporting the maintainer but will be developed in such a way that it could be extended to be used at any point during the development process.
7. Evaluation

Section 6 described a list of research questions to be addressed by the dissertation as well as the research agenda used to guide the research. During the course of this work we will answer these questions by performing additional maintenance studies. We will also develop the SEURAT system, described in sections 6 and 8, to serve as a test-bed for evaluating the answers.

We will perform the evaluation by conducting a user-study using the SEURAT system. In this study, the subjects will be given a set of maintenance tasks to perform with assistance from SEURAT which will display, capture, and inference over the rationale. The users will be trained to use SEURAT and encouraged to make use of both SEURAT and the rationale. SEURAT will be instrumented to record when rationale is used and captured.

After each subject has performed their tasks, two evaluations will be performed. The first evaluation is by the subject, the second by the examiner. In the subject examination, the subject will answer a survey about the tasks and about using SEURAT for assistance. The study will be designed to verify the answers to the following questions:

1. How can rationale be used to assist in software maintenance? Did the rationale uses supported by SEURAT help the user? Did they utilize them?

2. How can decisions be captured with enough specificity to be useful yet still general enough to allow for inferencing? Did the users feel that the rationale presented was detailed enough to describe the reasons behind decisions? Were they allowed to enter rationale that was detailed enough to describe their reasons? Did the inferencing work correctly?

3. Does rationale differ for different types of software modifications? Does the system allow the user to enter the rationale for the reason for modifications as well as the rationale for how they performed the modification?

4. Does maintenance rationale differ from original rationale? This question will be answered by both the results of the manual studies and by examining the rationale entered during the experiment.

5. Are there portions of the design or phases of the design process (Figure 1) where rationale capture would be more useful than others? Which parts of the rationale were utilized the most by the users?

6. What is the relationship between rationale collected at different phases? Is it only via the design artifacts? This will need to be answered prior to development of SEURAT. The user evaluation will ask if they feel that they were able to access the appropriate rationale when needed, including rationale that may have been collected at an earlier stage.

7. How can rationale changes be propagated? SEURAT will automatically check to see where changes in the rationale may produce inconsistencies with either existing rationale or possibly even earlier development artifacts. The users will be asked if SEURAT detected these inconsistencies and responded to them appropriately.
The second evaluation will be performed by the examiner. This will involve evaluating the quality of the work resulting from the experiment and how long it took to complete it. The SEURAT system will keep track of the time that it was in use to estimate the time to complete the experiment (the subject will be asked to provide this information as well). The quality evaluation may be performed by a third party rather than by the author of this dissertation to avoid bias.

If possible, the experiment subjects will be software developers with varying backgrounds. The survey approach was chosen rather than a comparison of maintenance with and without SEURAT to avoid issues with differing levels of expertise. Such a comparison would be extremely useful but would require a very large sample size in order to avoid bias due to different levels of expertise. The maintenance tasks are time consuming and it would be impractical to find large numbers of people willing to make the time investment required to participate.
8. Implementation

In this section, we describe the SEURAT implementation. Figure 26, given in section 6, shows the system architecture for SEURAT. The system consists of four major components:

- Argument Editor and Analyzer
- Rationale Repository
- Inference Engine
- Argument Ontology

In addition, there is the software development environment used by the developer to create the software artifacts (code, diagrams, and documentation). This is likely to be a heterogeneous collection of tools which may have little or no integration with each other. There may also be an additional problem-reporting component used to trigger the software modifications.

The SEURAT components will each be generated independently with well defined interfaces between them to maximize reusability. The following sections describe each component in more detail, as well as some of the tools proposed to implement them.

The initial version of SEURAT will be developed as a single user system. This is necessary in order to keep the focus on the research goals and lessen the implementation burden.

8.1 Argument Editor and Analyzer

The Argument Editor and Analyzer needs to perform the following functions:

- Allow entry of rationale associated with the development artifacts and store it in the Rationale Repository.
- Allow the user to select arguments from the Argument Ontology as part of entering rationale. This will involve providing an interface for browsing the levels of the ontology.
- Allow the user to add new arguments to the Argument Ontology when necessary.
- Allow the user to browse the rationale associated with a development artifact, both at the same level and at other phases of the design. This will also include viewing the rationale history to see what decisions were made earlier in the process.
- Allow the user to request evaluation/verification of the rationale and to display the results.

The Argument Editor provides the primary interface to the user. This will be a graphical user interface. For this reason, the Argument Editor and Analyzer will probably be developed using Java. Java is preferred over other languages, such as Visual Basic or Visual C++ because it supports multiple hardware platforms. It also has the advantage that
it is available free of charge. The choice of Java, however, depends on its ability to interface with the other tools used by the system. If this interface is not possible, a different language, or even a combination of languages, will be used.

8.2 Inference Engine

The Inference Engine will be used to perform a number of useful operations over the rationale to verify the correctness and completeness of the rationale and to evaluate the soundness of the design decisions (as captured in the rationale). The following functionality is planned for the Inference Engine, with more added if possible:

- Verification that the rationale is structurally complete. This includes checking that all decisions have selected alternatives.
- Evaluation of the rationale to ensure that selected alternatives have more support than those that were not selected.
- Checking for tradeoff consistency within the rationale.
- Determining the impact on the rationale of changing the importance of arguments.
- Ensuring that alternatives are not selected that were tried previously and rejected.
- Propagating the affects of rationale changes to the rest of the rationale.
- Determining the affects of rationale changes on the design.

There are a number of ways that this could be implemented. Some of these inferences were already implemented in the prototype system InfoRat, using CLIPS. Developing the Inference Engine as a rule-based system has the advantage that it is easy to extend. The disadvantage is that these systems are often slow. The language chosen needs to be one that can easily interface with the other components of the system.

8.3 Rationale Repository

The Rationale Repository stores both the current and previous versions of the rationale. There are a number of requirements for the repository. It should:

- Support the rationale representation developed for SEURAT,
- Associate the rationale with the development artifacts,
- Support queries to retrieve the rationale,
- Maintain a history of changes to the rationale and supports queries on that history.

The repository needs to respond to rationale retrieval queries. This could be implemented in a number of ways. InfoRat used CLIPS to read in rationale stored as CLIPS objects. Querying was also implemented using CLIPS. This worked well for small amounts of rationale but may not respond fast enough to queries on large amounts of rationale.

The need to track rationale changes suggests that it may be useful to employ a version control system such as the Concurrent Versioning System (CVS) but it is not clear how
that system could be interfaced with a program (as opposed to being interacted with directly by the user). A more flexible approach would be to use a DBMS to store the rationale data. The table structures would need to be designed to support the rationale representation developed. Multiple versions of the rationale could be implemented similarly to how they were implemented in InfoRat by storing old and new versions with fields indicating the relationships between the versions. There are a number of different DBMS systems available. One potential candidate would be MYSQL since it is freely available and runs on multiple platforms. Again, one consideration would be how easy it is to communicate with the repository from the other components in the system.

8.4 Argument Ontology

The Argument Ontology will contain a hierarchy of arguments that are available for use in the rationale. Each argument will have an importance associated with it. These importances will be project specific and will be provided by the developer. The requirements for the ontology are fairly simple: support for retrieval of arguments and update of arguments and argument importance. While less functionality is required than in the Rationale Repository, the Argument Ontology will probably be implemented using the same language/system.

8.5 Software Artifact Development Environment

In order for rationale capture to be feasible, it should be integrated as closely as possible with the actual development of the software that the rationale is collected for. Ideally, this would be integrated into a tool or tools that the designer uses. Development environments range from writing text documents using a text editor to using a CASE tool. For the Unified Process, the most popular tools (developed in consultation with the Unified Process developers) are those in the Rational Suite [Rational, 2000]. These include Rational RequisitePro for requirements management and Rational Rose for analysis, design, and initial code generation. The big advantage of using the Rational tools is that they support the development process. The disadvantages are that they are often difficult to use, it is unclear how easy it will be to integrate with them, and the tools themselves are quite expensive and may be difficult to make available to those evaluating the system.

Even if the Rational tools are chosen, some artifacts will need to be stored outside the toolset. For example, the user interface design is likely to be done using a more general drawing tool such as Power Point. The choice of tool will determine the granularity at which the rationale can be attached to the development artifacts. Ideally, the rationale for a single decision would be attached directly to the portions of the development artifacts that are affected by that decision. This would allow the most applicable rationale to be displayed when needed. If rationale were attached at a lower granularity, say to a file containing source code that implements many decisions, it will be much more difficult to determine which rationale goes with which portions of that code.

For example, if the user interface design is done in Power Point, the rationale for the design would be attached at the file level. If each display is created in a separate file that
will work but is not as useful as if the rationale could be attached to portions of the display. For code, regular text files will be used. In this case, the detail at which rationale can be attached will depend on how language-neutral SEURAT is. In order to attach rationale at a level lower than a file, SEURAT would need to be able to interpret the code to pick out information such as class and module names. This information could then map to the applicable portions of the rationale. The current plan is to make use of the Rational tools as much as is feasible and to supplement these tools with use of text editors and Power Point.
9. Contributions

In the course of this research, we plan to produce the following:

1. **A categorization of different ways to use DR during maintenance and what has to be done with the DR to support these uses**: this categorization is necessary in order to create a representation and system that supports maintenance. It is also necessary to avoid taking a “kitchen sink” approach to rationale collection where all reasons are captured even though some decisions may never need to be re-examined once made.

2. **A method for propagating changes made during maintenance through the rationale to ensure that it is kept current and to capture how the rationale evolves over time**: the software life-cycle is very long and it is highly likely that the design and rationale will continue to change. It is important to keep track of these changes and to ensure that decisions made for part of the design are still consistent with those made elsewhere.

3. **A representation for rationale that supports the following**:
   a. **Rationale occurring at multiple levels in the development process from requirements through maintenance**: this representation will support rationale for the requirements, rationale for the use cases and analysis classes, rationale for the class structure, and rationale for the code.
   b. **Rationale to support inferencing**: this representation will be structured as argumentation that can be used to support both semantic and syntactic inferencing. The rationale will contain enough detail to be useful yet will also be general enough so that contents can be compared.
   c. **Rationale to support maintenance**: this will include both the rationale for why changes are necessary as well as the rationale for how the changes were performed.

4. **A design rationale ontology that supports inferencing by indicating the relationships between arguments at different levels of abstraction**: this ontology will allow arguments to be captured in sufficient detail to explain the reasons behind the decisions yet also allow arguments to be compared at a more abstract level in order to evaluate the design.

5. **A way of attaching the rationale to the development artifacts (diagrams and code) so that it can be presented to and modified by the user**: one method for minimizing the intrusiveness of rationale capture is to integrate it as closely to the development process as possible. One way to do this is to integrate it into a development tool already used by the designer. This requires associating the design rationale with the development artifacts so that it can both be entered by the user and presented by the user later when the artifacts are updated.

6. **A prototype system that uses these methods to support the maintainer**: a prototype system will allow us to test out the representation and ensure that it supports the intended uses. The prototype system will allow verification that design rationale can be useful during maintenance.
10. Schedule

The proposed approach will be developed in four main stages:

1. **The investigation stage** will complete the additional studies needed to refine the representation and inferencing proposed for this work.
   a. *Additional maintenance studies* (January 2002 - February 2002) will involve performing the additional types of maintenance tasks described in Section 6.1.2. In particular, a study will be performed where an existing requirement is modified.
   b. *Argument ontology development* (March 2002) will involve developing the content and representation for the argument ontology.
   c. *Rationale representation development* (April 2002 - May 2002) will involve designing the representation for the design rationale.
   d. *COTS tool evaluation* (January 2002 - May 2002) will involve installing and evaluating candidate tools (such as Rational Rose) for use in implementing SEURAT.

2. **The implementation stage** will involve the building of the various components of the SEURAT system and integrating them together.
   a. *Rationale Repository development* (June 2002 - July 2002) will involve designing and implementing the Rationale Repository and the basic queries required to access the rationale.
   c. *Argument Editor and Analyzer development* (September 2002 - October 2002) will involve implementing the Argument Editor and Analyzer.
   d. *Inference Engine development* (November 2002 - December 2002) will involve implementing the Inference Engine.
   e. *System Integration and Test* (January 2003) will involve integrating the various components together and ensuring that they work correctly.

3. **The evaluation stage** will involve both setting up the evaluation experiment and conducting it.
   a. Rationale repository population (February 2003) will involve storing the rationale required for the experiment into SEURAT.
   b. *Experiment design* (March 2003) will involve creating the subject questionnaire and training materials.
   c. *Experiment conduction* (April 2003) will involve conducting the actual experiments with the research subjects.
   d. *Results evaluation* (May 2003 - June 2003) will involve evaluating the results of the experiment.

4. **The documentation stage** (April 2003 - August 2003) will involve writing up the results of the previous three stages into the final dissertation.
11. References


pany, New Jersey.


Lander, S.E., Lesser, V.R.: 1992, Customizing Distributed Search Among Agents with Heterogeneous Knowledge. Proc. 5th Int. Symp. on AI Applications in Manuf. & Robotics, Cancun, Mexico, December.

Osterweil, L.J.: 1987, Software Processes are Software Too, in *Proceedings of ICSE 9*, Monterey, CA, pp. 2-
13.


