AN INTEGRATED APPROACH FOR SOFTWARE DESIGN CHECKING USING DESIGN RATIONALE

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

Abstract. *Design Rationale* (DR), the reasons behind decisions made while designing, offers a richer view of both the product and the decision-making process by providing the designer's intent behind the decisions. DR is also valuable for checking to ensure that the intent was adhered to throughout the design, as well as pointing out any unresolved (or undocumented) issues that remain open. While there is little doubt of the value of DR, it is typically not captured during design. SEURAT (Software Engineering Using RATionale) is a system we have developed to explore uses of design rationale. It supports both the display of and inferencing over the rationale to point out any unresolved issues or inconsistencies. SEURAT is tightly integrated with a software development environment so that rationale capture and use can become integrated into the software development process.

1. Introduction

For a number of years, members of the Artificial Intelligence (AI) in Design community have studied *Design Rationale* (DR), the reasons behind decisions made while designing. DR offers a rich view of both the product and the decision-making process by providing the designer's intent behind the decisions. DR is also valuable for checking to ensure that the intent was adhered to throughout the design as well as pointing out any unresolved (or undocumented) issues that remain open.

An area where rationale for past decisions is especially useful is during software maintenance. One reason for this is that the software lifecycle is a long one. Large projects may take years to complete and spend even more

time out in the field being used (and maintained). Maintenance costs can be more than 40 percent of the cost of developing the software in the first place (Brooks, 1995). The panic over the "Y2K bug" highlighted the fact that software systems often live on much longer than the original developers intended. Also, the combination of a long life cycle and the typically high personnel turnover in the software industry increases the probability that the original designer is unlikely to be available for consultation when problems arise.

All these reasons argue for as much support as can be provided during maintenance. Semi-automatic maintenance support systems, such as Reiss's constraint-based system (Reiss, 2002), that work on the code, abstracted code, design artifacts, or meta-data, assist with maintaining consistency between artifacts. Design Rationale, however, assists with maintaining consistency in designer reasoning and intent.

1.1 DIFFICULTIES WITH RATIONALE

While rationale has great potential value, rationale is not in widespread use. One difficulty, despite much research, is the capture of design rationale. Recording all decisions made, as well as those rejected, can be time consuming and expensive.

Documenting the decisions can impede the design process if decision recording is viewed as a separate process from constructing the artifact (Fischer, et al., 1995). Designers are reluctant to take the time to document the decisions they did not take, or took and then rejected (Conklin and Burgess-Yakemovic, 1995). A real danger is the risk that the overhead of capturing the rationale may impact the project schedule enough to make the difference between a project that meets its deadlines and is completed, versus one where the failure to meet deadlines results in cancellation (Grudin, 1995). One way to mitigate these risks is to provide tools for rationale capture and use that are tightly integrated with those used during the designing process so that capturing and using the rationale becomes part of the standard process, not an extra task that needs to be performed with its own set of tools and standards.

1.2 USES OF RATIONALE

The key to making the capture worthwhile, as well as providing requirements for DR representation, is the use for, *and usefulness of*, the rationale. In this paper, we describe the SEURAT (Software Engineering Using RATionale) system, which integrates tools for rationale capture, visualization, and use into a standard software engineering environment. SEURAT addresses a number of uses for rationale:

- *Design verification* – using rationale to verify that the design meets the requirements and the designer's intent.
- *Design evaluation* – using rationale to evaluate (partial) designs and design choices relative to one another to detect inconsistencies.
- *Design maintenance* – using rationale to locate sources of design problems, to indicate where changes need to be made in order to modify the design, and to ensure that rejected options are not inadvertently re-implemented.
- *Design assistance* – using rationale to clarify discussion, check impact of design modifications, and perform consistency checking.

This paper is structured as follows: in section 2, we describe related work. In section 3, we describe the overall approach. Section 4 describes the rationale representation developed for SEURAT and section 5 presents the Argument Ontology, a key component of the rationale representation. Section 6 describes inferences to be performed over the rationale and section 7 gives the conclusions and outlines future work.

2. Related Work

Work on design rationale has focused on three main issues: capture, representation, and use. While SEURAT supports capture by providing the capability to enter the rationale, capture is not a main focus of the work. The related work on representation is presented as part of the representation discussion in section 4. In this section, we describe related work on rationale use.

2.1 RATIONALE USE

There are a number of systems that focused on uses for rationale for both engineering and software design. JANUS (Fischer, et al., 1995) critiques the design and provides the designers with rationale to support the criticism SYBIL (Lee, 1990) and InfoRat (Burge and Brown, 2000) both check that the rationale behind each decision is complete. C-Re-CS (Klein, 1997) performs consistency checking on requirements and recommends a resolution strategy for detected exceptions.

Co-MoKit (Dellen, et al., 1996) uses a software process model to obtain design decisions and causal dependencies between them. WinWin (Boehm and Bose, 1994) aims at coordinating decision-making activities made by various "stakeholders" in the software development process. Bose (Bose, 1995) defined an ontology for the decision rationale needed to maintain the decision structure. The goal was to model the decision rationale in order to support decision maintenance by allowing the system to determine the impact of a change and propagate modification effects. Chung, et al. (2000)

developed an NFR Framework that uses non-functional requirements to drive the software design process, producing the design and its rationale.

2.2 EVALUATING USEFULNESS

While the usefulness of rationale has not been studied in as much detail as the capture and representation, there have been some experiments performed. Field trials performed using itIBIS and gIBIS (Conklin and Burgess-Yakemovic, 1995) indicated that capturing rationale was found to be useful during both requirements analysis and design, and that the process also helped with team communication by making meetings more productive. Karsenty (1996) studied how DR could be used to evaluate a design. In this study, 50% of the designers' questions were about the rationale behind the design and 41% of those questions were answered using the recorded rationale.

3. Approach

For the SEURAT system we have chosen to focus our efforts on software engineering and concentrate on how rationale could be used during software maintenance, one of the most difficult and expensive phases of the software life cycle. Our goal is to create a system that can be tightly integrated with existing development tools so that rationale capture and use can become a part of the development process, not something that is done after the fact.

We are currently building the SEURAT system as a plug-in to the Eclipse Tool Platform (www.eclipse.org) so that it can be tightly integrated with a Java IDE (Interactive Development Environment) and other design tools that plug into Eclipse. This allows us to connect the rationale with the code and design artifacts that it explains. It ensures that the software maintainers are aware of and use the rationale.

SEURAT 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 inferences 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 modifications 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 modification.

Figure 1 shows SEURAT as part of the Eclipse Java IDE. SEURAT participates in the development environment in three ways: a Rationale Explorer (lower left pane) that shows a hierarchical view of the rationale and allows display and editing of the rationale; a Rationale Task List (lower right pane), that shows a list of errors and warnings about the rationale; and

Rationale Indicators that appear on the Java Package Explorer (upper left pane) and in the Java Editor (upper right pane) to show where rationale is available for a specific Java element. The examples in this paper come from a meeting scheduling system. Note that the screenshots are in color, making the icons much easier to distinguish than when reproduced in black and white.

The software developer enters the rationale to be stored in SEURAT while the system the rationale describes is being developed. SEURAT supports the entry by providing rationale entry screens for each type of rationale element.

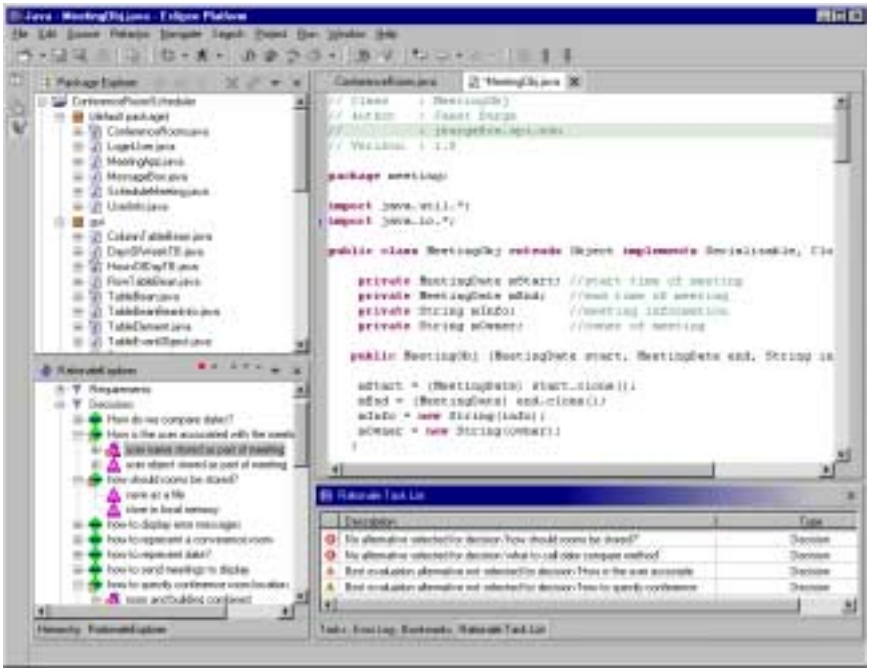

***Figure 1*. SEURAT and Eclipse**

4. Representation

A DR representation needs to be formalized and well structured, as opposed to just free text, so that computer-based checking and inferences are possible. We have generated a rationale representation, called RATSpeak, and have chosen to use a semi-structured argumentation format because we feel that argumentation is the best means for expressing the advantages and disadvantages of the different design options considered. Argumentation formats date back to Toulmin's representation (Toulmin, 1958) of datums, claims, warrants, backings and rebuttals. This is the origin of most

argumentation representations. More recent argumentation formats include Questions, Options, and Criteria (QOC) (MacLean, et al., 1995), Issue Based Information System (IBIS) (Conklin and Burgess-Yakemovic, 1995), and Decision Representation Language (DRL) (Lee, 1991). Each argumentation format has its own set of terms but the basic goal is to represent the decisions made, the possible alternatives for each decision, and the arguments for and against each alternative.

Argumentation has been used in rationale representations that were created specifically for 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 DRL, the language used in SIBYL. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale (Peña-Mora and Vadhavkar, 1996). This system is used to select design patterns based on the designers intent and other constraints.

We chose to base RATSpeak on DRL because DRL appeared to be the most comprehensive of the rationale languages. Even so, it was necessary to make some changes because DRL did not provide a sufficiently explicit representation of some types of argumentation (such as indicating if an argument was for or against an alternative).

Figure 2 shows the argumentation structure used in RATSpeak. The alternatives for each decision problem can be argued by their relationships to requirements, their relationships to other alternatives, and by assumptions or claims that support or deny the alternatives. The diagram also shows how decisions can be subdivided into sub-decisions and how selecting an alternative can result in additional decisions being needed.

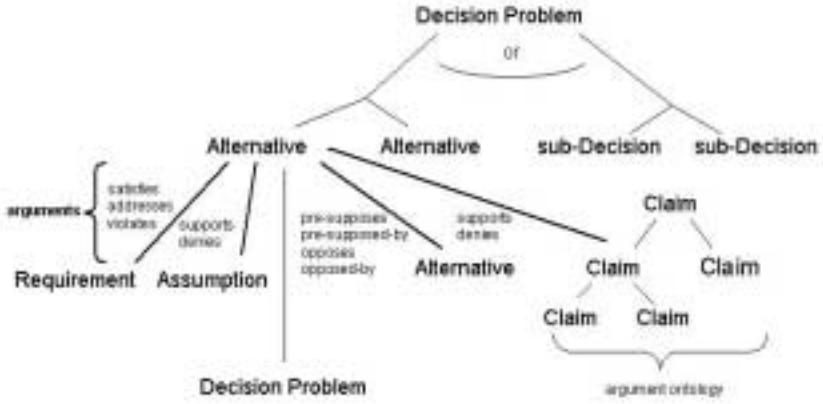

***Figure 2*. RATSpeak Argumentation Structure**

RATSpeak uses the following elements as part of the rationale:

- *Requirements* – these are the requirements, both functional and non-functional. These can either be represented explicitly in the rationale or be pointers to requirements stored in a requirements document or database. For the purposes of our examples, we will show them as part of the rationale. Requirements serve two purposes in RATSpeak, one is as the basis of arguments for and against alternatives. This allows RATSpeak to capture cases where an alternative supports or violates a requirement. The other purpose is so that the rationale for the requirements themselves can be captured.
- *Decision Problems* – these are the decisions that must be made as part of the development process. They are expressed in the form of questions.
- *Questions* – these are questions that need to be answered before the answer to the decision problem can be defined. The question can include the procedures or programs that need to be run or simple requests for information. While questions are not a standard argumentation concept, they can augment the argumentation by specifying the source of the information used to make the decisions, which is useful during software maintenance.
- *Alternatives* – these are alternative solutions to the decision problems. Each alternative will have a status that indicates if it is accepted, rejected, or pending.
- *Arguments* – these are the arguments for and against the proposed alternatives. They can either contain requirements (i.e., an alternative is good or bad because of its relationship to a requirement), claims about the alternative, assumptions that are reasons for or against choosing an alternative, or relationships between alternatives (indicating dependencies or conflicts). Each argument is given an *amount* (how much the argument applies to the alternative, i.e., how flexible, how expensive) and an *importance* (how important the argument is to the overall system or to the specific decision).
- *Claims* – these are reasons why an alternative is good or bad. Each claim maps to an entry in an *Argument Ontology* of common arguments for and against software design decisions. Each claim also indicates what *direction* it is in for that argument. For example, a claim may state that a choice is *NOT* safe or that an alternative *IS* flexible. This allows claims to be

stated as either positive or negative assertions. Claims also contain an *importance*, which can be inherited or overridden by the arguments referencing the claim.

- *Assumptions* – these are similar to claims except that it is not known if they are always true. Assumptions do not map to items in the Argument Ontology.
- *Argument Ontology* – this is a hierarchy of common argument types that serve as types of claims that can be used in the system. These are used to provide the common vocabulary required for inferencing. Each ontology entry contains an *importance* that can be overridden by claims that reference it.
- *Background Knowledge* – this contains *Tradeoffs* and *Co-Occurrence Relationships* that give relationships between different arguments in the Argument Ontology. This is not considered part of the argumentation but is used to check the rationale for any violations of these relationships.

Figure 3 shows the relationships between the different rationale entities.

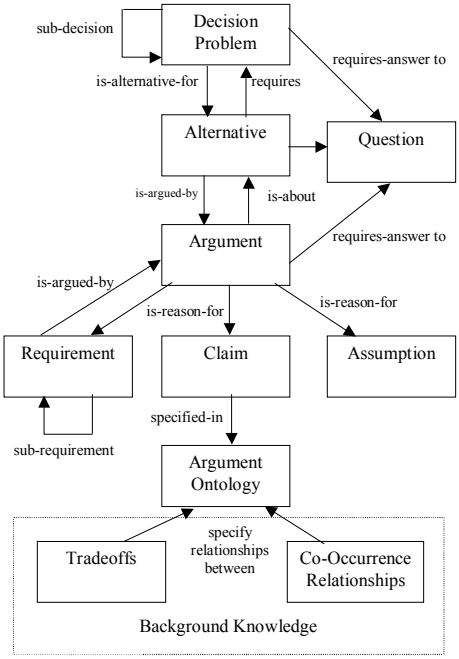

***Figure 3*. Relationships Between Rationale Entities**

5. Argument Ontology

One key element in the RATSpeak representation is the Argument Ontology. Our work on InfoRat showed the importance of providing a common vocabulary to support inferencing over the content of the rationale as well as over its structure. To support this, we have developed an ontology of reasons for choosing one design alternative over another. This ontology forms a hierarchy of terms with abstract reasons at the root and increasingly detailed reasons towards the leaves.

RATSpeak provides the ability to express several different types of arguments for and against alternatives. One type of argument is if an alternative satisfies or violates a requirement. Other arguments refer to assumptions made or dependencies between alternatives. Another type of argument involves claims that an alternative supports or denies a Non-Functional Requirement (NFR). These NFRs, also known as "ilities" (Fillman, 1998) or quality requirements, refer to overall qualities of the resulting system, as opposed to functional requirements, which refer to specific functionality. As we describe in (Burge and Brown, 2002), the distinction between functional and non-functional is often a matter of context. RATSpeak also allows NFRs to be represented as explicit requirements.

There have been many ways that NFRs have been organized. CMU's Quality Measures Taxonomy (SEI, 2000) organizes quality measures into Needs Satisfaction Measures, Performance Measures, Maintenance Measures, Adaptive Measures, and Organizational Measures. Bruegge and Dutoit (2000) break a set of design goals into five groups: performance, dependability, cost, maintenance, and end user criteria. Chung, et al. (2000) provides an unordered list of NFRs as well as specific criteria for performance and auditing NFRs.

For the RATSpeak argument ontology, we took a bottom-up approach by looking at what characteristics a system could have that would support the different types of software qualities. This involved reviewing literature on the various quality categories to look for how a software system might choose to address these qualities. For example, one quality attribute that is a factor in design decisions is scalability. We looked to see what might contribute toward scalability in a software design and added these attributes to the ontology. For example, one way to increase scalability is to minimize the number of connections a system must set up, another is to avoid using fixed data sizes that may limit the capacity of the system. Our aim was to go beyond the idea of design goals or quality measures to look at *how* these qualities might be achieved by a software system.

In maintenance, the maintainers are more likely to be looking at the lower-level decisions and will need specific reasons why these decisions

contribute to a desired quality of the overall system. It is probable that decisions made at the implementation level are likely to correspond to detailed reasons in the ontology, while higher level decisions are more likely to use reasons at the more abstract levels.

After determining a list of detailed reasons for choosing one alternative over another, an Affinity Diagram (Jiro, 2000) was used to cluster similar reasons into categories. These categories were then combined again. The more abstract levels of the hierarchy were based on a combination of the NFR organization schemes listed earlier (the CMU taxonomy as well as Bruegge and Dutoit's design goals). Also, NFRs from the Chung list were used to fill in gaps in the ontology.

Figure 4 shows the first two levels of the Argument Ontology displayed in SEURAT.

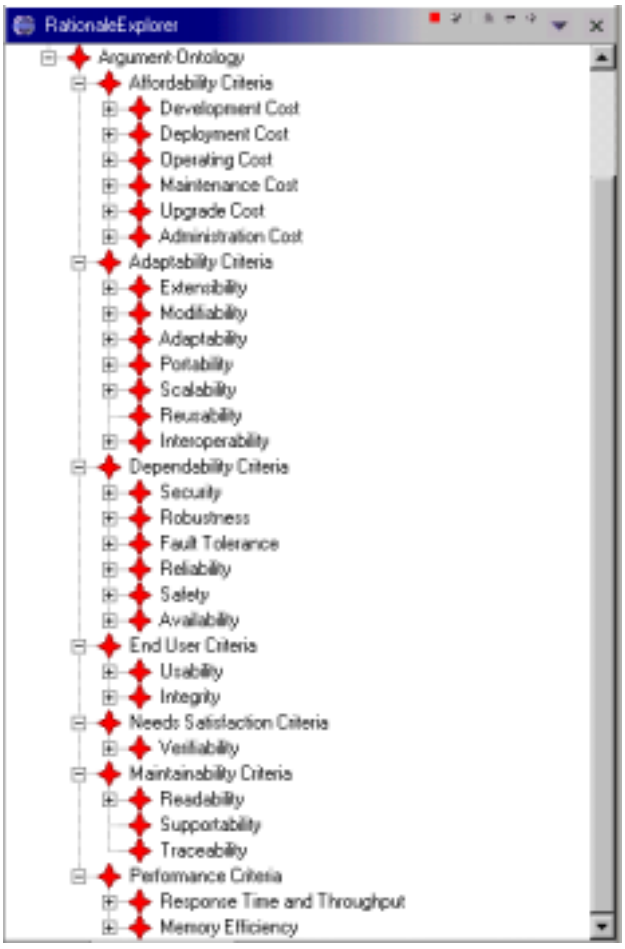

***Figure 4*. Top Levels of Argument Ontology**

Each of these criteria then have sub-criteria at increasingly more detailed levels. As an example, Figure 5 shows the sub-criteria for Usability as displayed in SEURAT. The ontology terms are worded in terms of arguments: i.e., <*alternative*> is a good choice because it <*ontology entry* >, where *ontology entry* starts with a verb. The SEURAT system has been designed so that the user can easily extend this ontology to incorporate additional arguments that may be missing. With use, the ontology will continue to be augmented and will become more complete over time. It is possible to add deeper levels to the hierarchy but that will make it more time consuming for the developer to find the appropriate item when adding rationale.

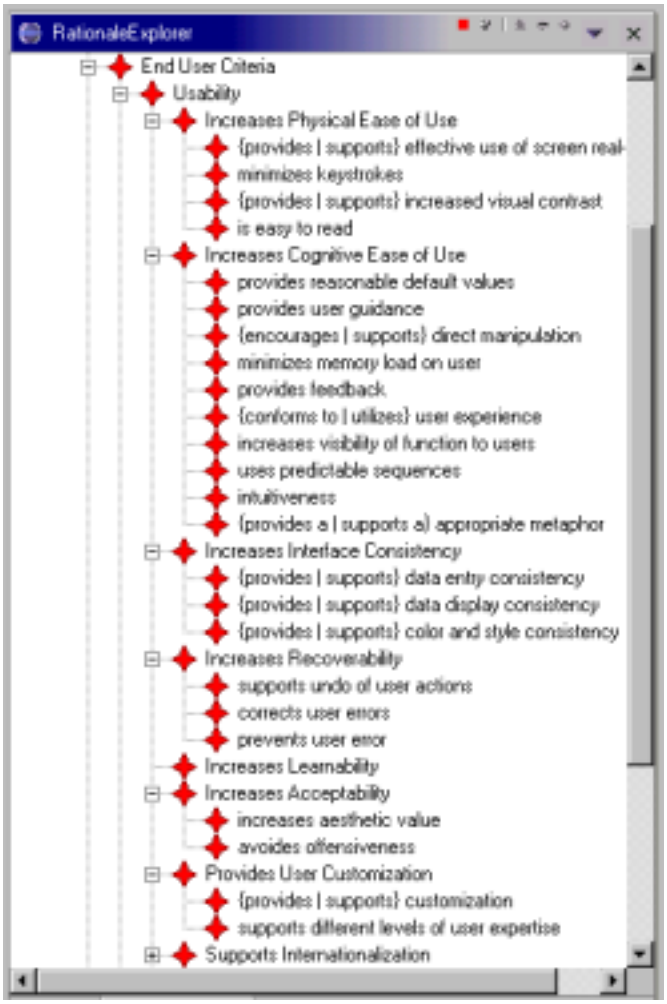

***Figure 5*. Sub-Criteria for Usability**

Similar hierarchies have been developed for the other categories in Figure 4. One thing to note is that it is not a strict hierarchy—there are many cases where items contributing toward one quality also apply to another. One example of this is the strong relationship between scalability and performance. Throughput and memory use, while primarily thought of as performance aspects, also impact the scalability of the system. In this case, and others that are similar, items will belong to more than one category.

The argument ontology also includes a default importance for each item. These are present so that SEURAT users can specify this information for a particular system. This is used in weighing the different arguments during inferencing. The importance can be overridden for each claim or argument but is stored with the ontology to allow this information to be global if desired.

Other relationships that need to be captured are tradeoffs and co-occurrences. These are cases where two items in the ontology often either oppose each other in arguments or support each other in arguments. For example, avoiding variable re-use makes it easier to verify the software is correct but also means the program may take more memory. The user can represent this, and similar tradeoffs, as background knowledge stored as part of the rationale. This background knowledge refers to the items in the argument ontology and stores the relationships between them.

6. Support for Rationale Use

Design Rationale is very useful even if it is only used as a form of documentation that provides extra insight into the designer's decision-making process. SEURAT supports the viewing of DR by allowing the software developer to associate the rationale with the code and by using Rationale Indicators to show which pieces of code have rationale available. Figure 6 shows a portion of the Package Explorer from the Eclipse Java IDE where the presence of rationale is indicated by a small modification to the upper left hand corner of the "J" icon indicating a Java file.

DR can provide even more useful information about the design and modifications made to the design if there is a way to perform inferences over it. Due to the nature of DR, the results may be in the form of warnings or information (as opposed to conclusions) that help the developer keep track of the development process and help the maintainer act carefully and consistently. In the following sections we describe a number of different SEURAT inferences both implemented and planned.

We have chosen to break our inferences into four categories: syntactic, semantic, queries, and historical. Syntactic inferences are those that are concerned mostly with the structure of the rationale. They look for information that is missing. Semantic inferences require looking into the

content of the rationale to evaluate the consistency of the design reasoning and point out cases made where less-supported decisions were made. Rationale queries give the user the ability to ask questions about the rationale, and historical inferences use a history of rationale changes to help the user learn from past mistakes, rather than repeating them.

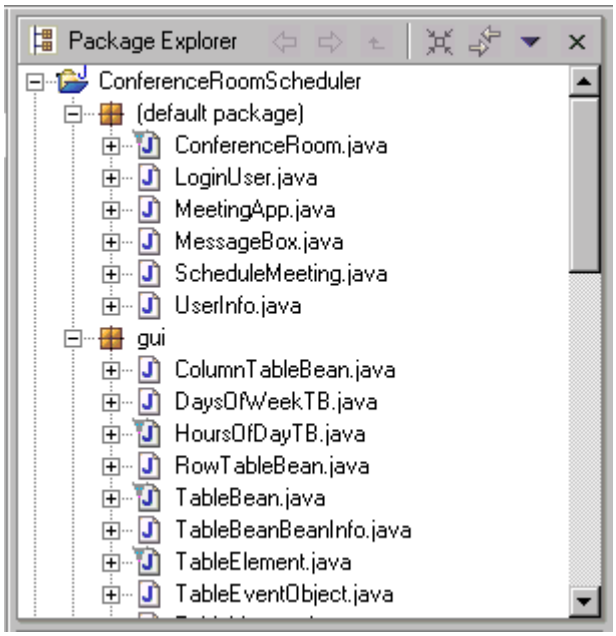

Figure 6. **Package Explorer Showing Rationale Associations**

6.1 SYNTACTIC INFERENCING

Syntactic inferencing is primarily concerned with the structure of the rationale – ensuring that the rationale is complete. This is a significant aid to the developer to make sure they do not leave any unresolved issues behind when building the system. These inferences include the following:

- Checking for decisions with no selected alternatives;
- Checking for decisions with more than one selected alternative when there should be only one;
- Checking for selected alternatives with no arguments in their favor;
- Checking for selected alternatives with only arguments in opposition;
- Checking for biased arguments where some alternatives have many arguments (for and/or against) while others have few or none.

SEURAT currently displays which alternative has been selected for each decision and indicates that there is an error if no alternative is selected.

Errors are shown in two places: an error indicator on the rationale item in the Rationale Explorer and an error description on the Rationale Task List. Figure 7 shows the Rationale Explorer with an error indicator showing that no alternative was selected for the decision "what to call date compare method" and Figure 8 shows the Rationale Task List that displays that error, and others. Errors are indicated by a red icon containing an "X".

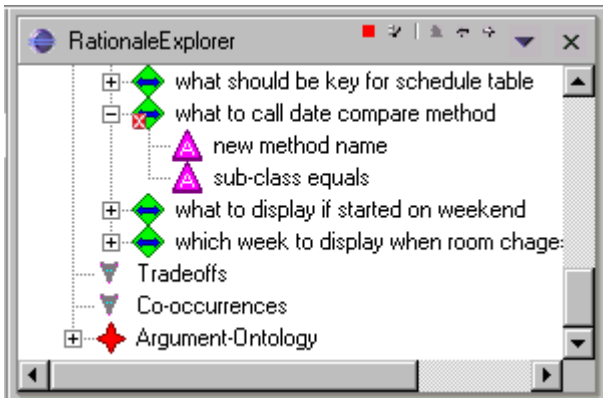

***Figure 7*. Rationale Explorer Showing Rationale Error**

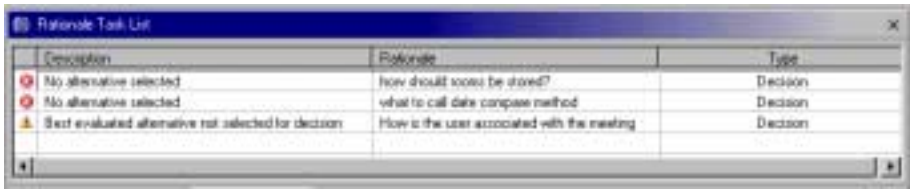

***Figure 8*. Rationale Task List**

6.2 SEMANTIC INFERENCES

While syntactic inferences look at the structure of the rationale, the semantic inferences look at the content. This allows a more in-depth look for any inconsistencies in reasoning that are captured in the rationale. The syntactic inferences implemented in SEURAT include the following:

- Determining if the best supported alternatives were selected;
- Checking for contradictory arguments by using the argument ontology to compare claims;
- Checking for violated requirements;
- Checking for violations of the tradeoff and co-occurrence relationships captured in the rationale.

Some of these results are shown as errors, such as when a requirement is violated, while others are warnings. Figure 9 shows how a warning is

indicated when the inferencing shows that the best alternative was not selected for the decision "How is the user associated with the meeting." This warning also shows up on the Rationale Task List shown in Figure 8. The warning is displayed as a yellow triangle icon with an exclamation point inside it.

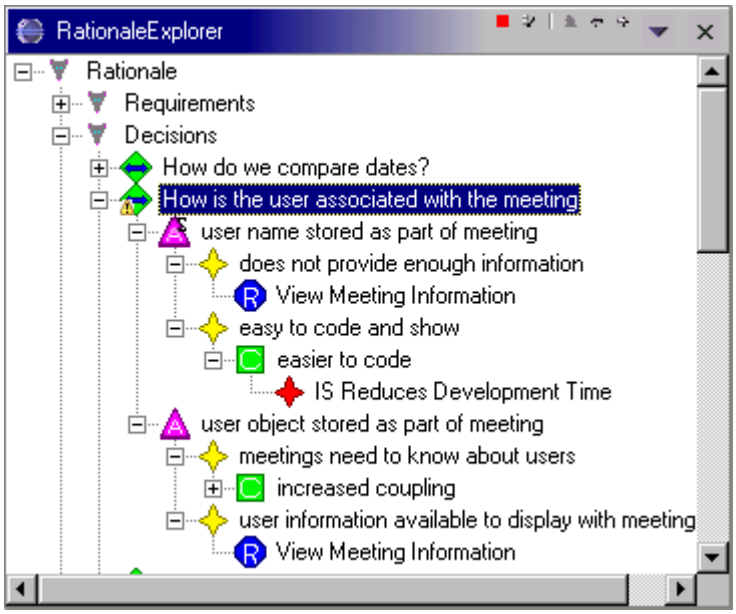

***Figure 9*. Rationale Explorer with Warning Indicator**

The semantic inferencing also allows the user to do some "what-if" reasoning by making changes to the rationale and seeing what effect that has on the decisions that have been made. For example, SEURAT provides the ability to disable requirements or assumptions and re-compute the evaluation of each decision. Figure 10 shows the Rationale Explorer after an assumption, "customer normally combines room and building" has been disabled. The assumption, denoted by an icon containing an "A", is changed to have a "D" in the upper right hand corner showing it is disabled. When the decision is re-evaluated, a warning icon is shown because the selected alternative (denoted by an "S" in the upper right hand corner) is no longer the best supported. The new warning is added to the Rationale Task List shown in Figure 11.

Another way that semantic inferencing is useful is in evaluating the effect of changing priorities on the design. Arguments for and against alternatives can consist of requirements, other alternatives (in case of dependencies), arguments, and claims. Each argument has an importance

associated with it that can either be set at the argument level of, in the case of assumptions and claims, inherited. Each claim is associated with an entry in the argument ontology, which also has an importance assigned. The user can change the importance at any of the three levels (ontology, claim, or argument) and will be able to examine how that change affects the evaluation of the rationale.

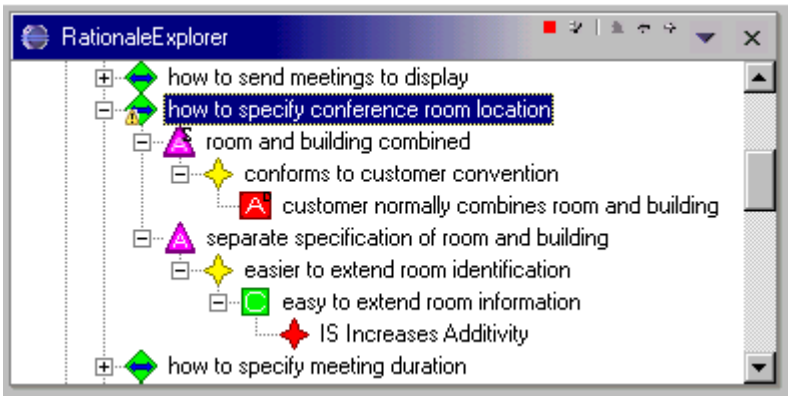

Figure 10. **Rationale Explorer with Disabled Assumption**

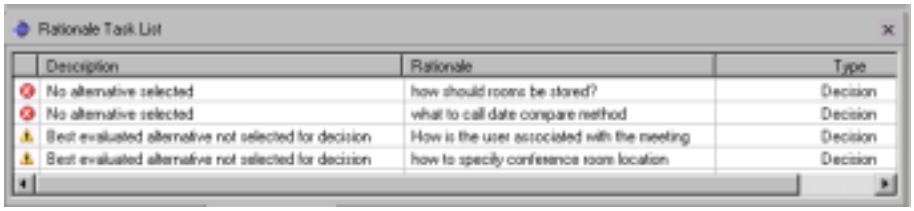

Figure 11. **Rationale Task List with New Warning**

6.3 RATIONALE QUERIES

Rationale queries are inferences that are performed only upon request, not automatically when the rationale changes. These queries are used to obtain additional information about the rationale or the design. Rationale queries supported by SEURAT include the following:

- Listing all selected alternatives that address or satisfy a specific requirement;
- Listing all non-selected alternatives that address or satisfy a specific requirement;
- Listing which alternatives are argued (for or against) by a specific claim or specific ontology entry;

- Listing where there were importance overrides (from the default specified in the argument ontology) in the rationale;
- Listing all disabled entities (assumptions, claims, or requirements) in the rationale;
- Listing the most frequently referenced ontology entries (i.e., common arguments for and against alternatives).

The results of these queries will not result in errors or warnings about the rationale but will provide useful information to assist the developer in understanding the rationale and the system it describes.

6.4 HISTORICAL INFERENCING

The final component of SEURAT inferencing is inferences that take advantage of stored history about changes to the rationale. One of the motivating reasons for keeping track of rationale is to avoid repeating past mistakes by documenting alternatives that were attempted and then rejected. History-based inferences would be used to ensure that the developer does not select an alternative that was rejected before without being aware of the reasons for why it failed the first time. The rationale history is also used to keep track of which areas of the design have been the most volatile.

6.5 SEURAT IMPLEMENTATION

SEURAT is implemented as a Java Plugin for the Eclipse framework. This provides tight integration with the Eclipse Java IDE where the rationale associations are shown as part of the Java editor used to develop the code that the rationale describes and where the rationale and rationale status displays are all shown as windows within the IDE. The rationale is stored in a MySQL database. This provides scalability to large amounts of rationale and allows the use of SQL queries to assist in the inferencing. The database relationships can be used to propagate the results of rationale changes to other affected portions of the rationale. These links between the rationale act much like dependencies described in a truth maintenance system except that we are not asserting the truth of the statements.

7. Conclusions and Future Work

The SEURAT system provides a number of important innovations contributing towards effective use of rationale for software maintenance. The first is the argument ontology. This contributes in several ways: first, by creating an extensive list of reasons for making software design choices and secondly by using these reasons to support semantic inferencing to determine the impact of these choices on the software system and to promote consistency in the rationale. Another key contribution is the

integration of SEURAT into a software development environment used by the developers and maintainers. This allows both the developers and maintainers to use the rationale without having to remember to invoke a separate utility or environment and lessens the disruption that can occur when switching from development to documentation.

Future work on SEURAT will involve expansion of the inference set and enhancements to the integration with the Eclipse Java IDE. These changes will increase both the functionality and usability of the SEURAT system. The system will be evaluated in a series of experiments with software developers of varying levels of expertise performing a series of maintenance tasks to determine the effectiveness of the rationale support.

We feel that the SEURAT system will be invaluable during development and maintenance of software systems. During development, SEURAT will help the developers ensure that the systems they build are complete and consistent. During maintenance, SEURAT will provide insight into the reasons behind the choices made by the developers during design and implementation. The benefits of DR are clear but only with appropriate tool support, such as that provided by SEURAT, can DR live up to its full potential as an aid for revising, maintaining, and documenting the software design and implementation.

References

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

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

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

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

Burge, JE and Brown, DC: 2002, *NFRs: Fact or Fiction?*, Technical Report WPI-CS-TR-02-01, Computer Science Department, WPI.

Bruegge D and Dutoit A: 2000, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems,* Prentice Hall.

Chung, L, Nixon, BA, Yu, E, and Mylopoulos, J: 2000, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers.

CMU: 2002, Quality measures taxonomy, http://www.sei.cmu.edu/str/taxonomies/view_qm.html

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

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

Filman, RE: 1998, Achieving Ilities, in *Proc. of the Workshop on Compositional Software Architectures*, Monterey, CA, USA.

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

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

Jiro, K: 2000, *KJ Method: A Scientific Approach to Problem Solving*, Tokyo: Kawakita Research Institute.

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

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

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

Lee, J: 1991, Extending the Potts and Bruns Model for Recording Design Rationale, in *Proc. of the 13th International Conf. On Software Engineering*, Austin, TX, pp. 114-125.

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

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

Potts, C and Bruns, G: 1988, Recording the Reasons for Design Decisions, in *Proc. of the International Conf. On Software Engineering*, Singapore, pp. 418-427.

Reiss, SP: 2002, Constraining Software Evolution, in *Proc. of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, pp. 162-171.

Toumlin, S: 1958, *The Uses of Argument*, Cambridge University Press