13 Rationale-Based Support for Software Maintenance

J.E. Burge, D.C. Brown

Abstract: One of the many difficulties encountered while performing software maintenance is determining the impact of potential changes on what already exists. One way to address this difficulty is to give the maintainers access to the Design Rationale of the original system. This rationale would provide the intent behind the design and implementation decisions, as well as a history of design alternatives that have been considered. Unfortunately, this information is difficult and time consuming to capture and therefore is rarely available. Our approach to this problem is to look at how the rationale could be used. Rationale needs to be useful to provide incentive for its initial capture. We present SEURAT, a system that supports entry and display of the rationale as well as inferences over the rationale. It helps ensure that the reasoning given for modifications made during software maintenance is consistent with the designer's initial intent.

Keywords: design rationale; software maintenance; inference, argumentation

13.1 Introduction

Modifying working software that is currently in use is always a risky endeavor. It is very difficult to determine the impact of potential changes on what already exists. The problem gets even worse if the original developers of the software are not available or if the maintenance is turned over to an organization that did not initially design and build the software.

Some of these risks could be mitigated if the maintainers had access to the Design Rationale (DR) of the original system. DR documents the decision making process by capturing the intent behind the design and implementation decisions as well as a history of design alternatives that had been considered. The DR would include any assumptions made when developing the initial system and how they impacted the design. Assumptions can become invalid over time, which is a key reason for why software needs to continually evolve [19]. It is important to re-examine assumptions during maintenance to ensure that they still hold.

Unfortunately, most developers do not capture the rationale behind their decision-making. Recording rationale is seen as being time-consuming and disruptive. Documenting the decisions can impede the design process if

decision recording is viewed as a separate process from constructing the artifact [14]. Developers are also reluctant to document their mistakes by keeping track of what they tried that did not work and are concerned about potential liability if a decision they record becomes responsible for a catastrophic failure of the system [9]. Another issue is that once the rationale is captured, will it be *used* and how, exactly, will it be *useful*?

We have chosen to address the use of and usefulness of rationale because the use of the rationale is what is ultimately needed to motivate its capture. Rationale has many potential uses throughout the software development cycle. At each stage of the process, it is useful to know the reasons behind the decisions made earlier. In addition, the act of recording the rationale can encourage developers to investigate alternative solutions and can support their selections with arguments. We feel that rationale is especially useful, however, during the maintenance phase. Rationale is valuable because even if the original developers of the system are available they may not remember all the details behind each decision made during a process that could span years. The usefulness, of course, is still bounded by what rationale has been captured and developers may still be reluctant to record all reasons for their decisions. We feel, however, that the value of the rationale outweighs its cost and that developing compelling uses for rationale is an important step towards motivating the developers to record it.

To investigate the uses of rationale, we have developed a prototype system, SEURAT (Software Engineering Using RATionale) [6], that provides both retrieval of, and inferencing over, rationale. The main focus has been on developing uses that support maintenance but SEURAT could be used during other development phases [12] as well. In this chapter, we will summarize our research into how rationale can be used to support software maintenance and how that can be done using the SEURAT system.

The remainder of the chapter is structured as follows: Sect. 13.2 describes related work, Sect. 13.3 describes how rationale can be used during several types of software maintenance, Sect. 13.4 describes the SEURAT system and how it represents, captures, presents, and inferences over the rationale, Sect. 13.5 summarizes the SEURAT evaluation, and Sect. 13.6 concludes the paper and describes future work.

13.2 Related Work

There has been significant work on capture, representation, and use of design rationale in the field of engineering design. Lee [18] has written an excellent survey of this work. The use of rationale for software

development has been surveyed by Dutoit and Paech [11]. Potts and Bruns [25] created a model of generic elements in software design rationale that was then extended by Lee [17] to create Decision Representation Language (DRL), the basis of the RATSpeak representation used by SEURAT. Design Recommendation and Intent Model (DRIM) was used in a system to augment design patterns with design rationale [24]. This system is used to select design patterns based on the designers' intent and other constraints. WinWin [1] aims at coordinating decision-making activities made by various "stakeholders" in the software development process. Bose [2] defined 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. [10] developed an NFR Framework that uses nonfunctional requirements to drive the software design process, producing the design and its rationale.

There are also other systems that perform consistency checking. C–Re–CS [16] performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. Reiss [26] has developed a constraint-based, semi-automatic maintenance support system that works on the code, abstracted code, design artifacts, or meta-data to assist with maintaining consistency between artifacts.

Lougher and Rodden [21] investigated maintenance rationale and built a system that attaches rationale to source code. Their approach differs from ours, however, in that they argue that maintenance rationale is very different from that captured during development and is not in the form of argumentation. Canfora et al. [7] also address maintenance rationale and break rationale into two parts: rationale in the large (rationale for higher level decisions in maintenance) and rationale in the small (rationale for change and testing). The focus on the rationale in the small is on *how* the change will be implemented but does not appear to focus on reasons behind implementation choices at a low level. They developed the Cooperative Maintenance Conceptual Model (CM^2) which is based on the QOC [22] argumentation format.

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 [9] 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 [15] 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. Bratthall et al. [3]

performed an experiment using rationale to assist in performing changes on two different systems. For one system, rationale was shown to be helpful in decreasing the time used to make the changes and improving the correctness of the changes but results were inconclusive for the second system.

13.3 Rationale for Software Maintenance

To determine how rationale can be used in software maintenance it is useful to look at what types of maintenance might be performed. There are a number of different classifications for types of software maintenance [8]. The three types that we address are corrective, adaptive (a combination of four of Chapin's types), and enhancive. We chose these types because they affect modifications to the source code.

Corrective maintenance involves correcting failures of the system [20]. Rationale can be useful in detecting the source of some types of failures. For example, if a failure occurs because an assumption is no longer valid, the rationale may, in some cases, help detect what parts of the design and code depend on the assumption being true. This would point out some places where changes are likely to be necessary. The rationale may also contain some possible alternatives that might be better candidates and, if selected, could fix the problem. It can also indicate if there are alternatives that should be avoided. Rationale can also be used proactively to find problems that may not have appeared yet – if a failure points to a decision made earlier it might be advisable to look at the reasons for making the decision and see if these reasons were important in other choices as well. This could indicate areas that might need changes to avoid future failures.

Adaptive maintenance involves making changes to the system that do not change the functionality seen by the customer. This is a combination of four of Chapin's types: groomative (improving elegance or security), preventive (improving maintainability), performance (improving performance), and adaptive (changing to account for different technology or resource use) [8]. The rationale can provide a guide to where improvements should be made. There are likely to be cases where decisions were made for reasons that are important in the short term but may require revision in the future. For example, a developer may choose the alternative that will be the fastest and easiest to code even though it may not be as desirable as other alternatives. The rationale can be used to look for decisions made for the sake of expediency and show what some of the better alternatives are so that the developers can consider those when updating the code.

Enhancive maintenance involves replacing, adding, or extending "customer-experienced functionality" [8]. It is important to ensure that the reasons used to make the enhancements are consistent with those used while developing the existing system. For example, if performance was important in the initial system, it should also be considered important when adding new functionality. It would be unfortunate if the new design choices resulted in significantly slower response time. Rationale can be used to check for any tradeoff violations that might be made by new additions to the system. A tradeoff violation would occur if there were system attributes where more of one meant less of another (such as flexibility versus development time) and the maintainer only considered one of the attributes when making a decision. Rationale can also be used to evaluate the strength of new design alternatives based on priorities set when the initial system was developed.

In all types of maintenance it is critical to "do no harm" to the working system. Rationale can be used to capture dependencies between the different alternatives considered. This would prevent developers from spending time implementing an alternative that is incompatible with earlier design choices.

13.4 The SEURAT System

We have developed the SEURAT system to support the use of rationale to assist with software maintenance. SEURAT presents the rationale to the maintainer and inferences over it to detect problems and inconsistencies within the rationale that may also indicate problems with the design. SEURAT also supports capture of rationale and fits into the RMS framework [12] by providing a Rationale Capture Component, a Rationale Retrieval Component, and a Rationale Representation Component (with the former two dependent on the latter). 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 additional that is performed retrospectively after development is complete.

We have built the SEURAT system as a plug-in to the Eclipse Tool Platform (www.eclipse.org) so that it can be tightly integrated with an Interactive Development Environment (IDE). This allows us to connect the rationale with the code that it explains. This connection ensures that the software maintainers are aware of and use the rationale. The rationale is stored in relational database tables using MySQL.

SEURAT presents the relevant DR when it is needed and allows entry of new rationale for the modifications. The new DR will then be verified against the original DR to check for inconsistencies. There are two main types of checks that are made: structural inferences to ensure that the rationale is complete, for the decisions recorded, and evaluation, to ensure that the rationale is based on well-founded arguments. Of course, there is no way to ensure that all the designers' reasoning is recorded but SEURAT can check for omissions such as failing to select an alternative or selecting an alternative without any argumentation given.

Figure 13.1 shows SEURAT as part of the Eclipse Java IDE. SEURAT participates in the development environment in three ways: a Rationale Explorer (upper left pane), that shows a hierarchical view of the rationale and allows display and editing of it; 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 (lower left pane) and in the Java Editor (upper right pane), to show whether rationale is available for a specific Java element. The examples in this chapter come from a conference room scheduling system. Note that the screenshots are in color, making the icons much easier to distinguish on the actual SEURAT displays than when reproduced here in black and white.

This display design, which also reflects the architecture of the system, was chosen because it very closely parallels the Eclipse Java IDE. For example, the Rationale Explorer uses a tree view similar to that provided by the Java Package Explorer where items in the tree can be brought up in an editor by double-clicking on them. This tree format is also an appropriate one for showing the rationale argumentation and provides a high-level view of the rationale where the maintainer can choose how deep into the argumentation structure they want to go by "expanding" the rationale elements much like they would expand the view of Java files to show attributes and methods. The Rationale Task List was designed to be similar in appearance to the Tasks display provided by Eclipse. The Tasks display shows compilation errors and warnings about problems in the code while the Rationale Task List shows errors and warnings about problems in the rationale. The Java Editor used in SEURAT is the same as the one used in Eclipse. The Bookmark display is also the same except that SEURAT has added associations between alternatives in the rationale and elements in the code (files, classes, attributes, or methods) to the list. The maintainer can find the code mentioned in the bookmark by clicking on it and can find the rationale associated with code shown in the editor by moving their mouse cursor over the bookmark that indicates that rationale is present.

The software developer enters the rationale to be stored in SEURAT while the software system the rationale describes is being developed.

SEURAT supports this by providing rationale entry screens for each type of rationale element.

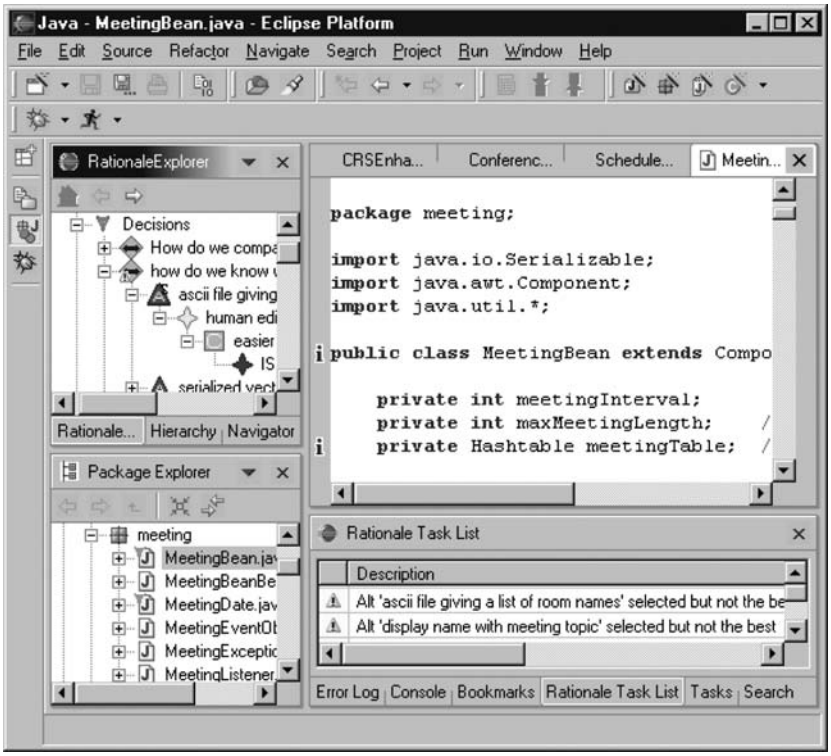

Fig. 13.1. SEURAT and Eclipse

SEURAT performs two main types of inferences over the rationale: syntactic inferences, which are concerned mostly with the structure (such as looking for missing relationships), and semantic inferences, which look at the content (such as evaluating the choices made). When problems are detected, they are displayed in two places: in the Rationale Explorer, as error and warning icons on the rationale, and on the Rationale Task List, which gives a more detailed explanation of what the problem is.

In the following sections, we will describe a subset of the capabilities provided by SEURAT and describe their use during software maintenance. The examples come from the rationale for a conference room scheduling system.

13.4.1 Representation

Before describing how rationale can be used, we first need to explain what our rationale contains. 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 an argumentation format because we feel that argumentation is the best means for expressing the advantages and disadvantages of the different design options considered.

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.

We have based RATSpeak on Lee's Decision Representation Language (DRL) [17] because DRL appeared to be the most comprehensive of the rationale languages and was designed to capture rationale for software design. 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).

RATSpeak uses the following elements as part of the rationale:

- *Requirements* – these include both functional and nonfunctional requirements. They can either be represented explicitly in the rationale or be pointers to requirements stored in a requirements document or database. Requirements serve two purposes in RATSpeak. One is as the basis of arguments for or against alternatives. This allows RATSpeak to capture cases where an alternative satisfies 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.
- *Questions* – these are questions that need to be answered before the answer to the decision problem can be defined. A question can include the procedures or programs that need to be run or who should be asked to get the answer. Questions augment the argumentation by specifying the source of the information used to make the decisions (the procedure, program, or person).
- *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 refer to 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, e.g., 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 or 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 or whether they will continue to hold in the future. 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 (e.g., Development Cost; Portability). These are used to provide the common vocabulary required for inferencing. Each ontology entry contains a default importance that can be overridden by claims that reference it. These arguments are tailored to the software development domain. A complete list of ontology entries can be found in Burge [4].
- *Background Knowledge* – this contains Tradeoffs and Co-Occurrence Relationships that give relationships between different arguments in the Argument Ontology. This is not the considered part of the argumentation but is used to check the rationale for any violations of these relationships.

Figure 13.2 shows the relationships between the different rationale entities.

RATSpeak provides the ability to express several different types of arguments for and against alternatives. One type of argument is that an alternative satisfies or violates a requirement. Other arguments refer to assumptions made or dependencies between alternatives. A fourth type of argument involves claims that an alternative supports or denies a Non-Functional Requirement (NFR). These NFRs, also known as “ilities” [13] 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 [5], the distinction between functional and nonfunctional

is often a matter of context. RATSpeak also allows NFRs to be represented as explicit requirements.

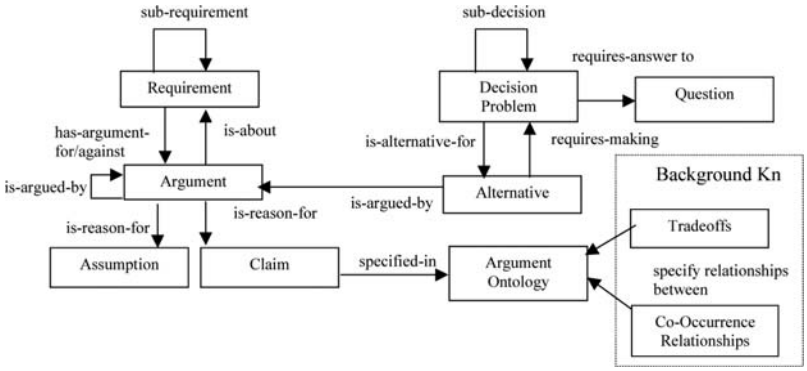

Fig. 13.2. Relationships between rationale entities

The RATSpeak representation describes the NFRs as part of the Argument Ontology. The Argument Ontology is a hierarchy of reasons for choosing one design alternative over another with abstract reasons at the root and increasingly detailed reasons towards the leaves. This is needed to provide a common vocabulary to support inferencing over the content of the rationale in addition to over its structure.

Figure 13.3 shows the top level of the Argument Ontology displayed in SEURAT.

Fig. 13.3. Top level of argument ontology

Each of these criteria then has subcriteria at increasingly more detailed levels. As an example, Fig. 13.4 shows some of the subcriteria 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.

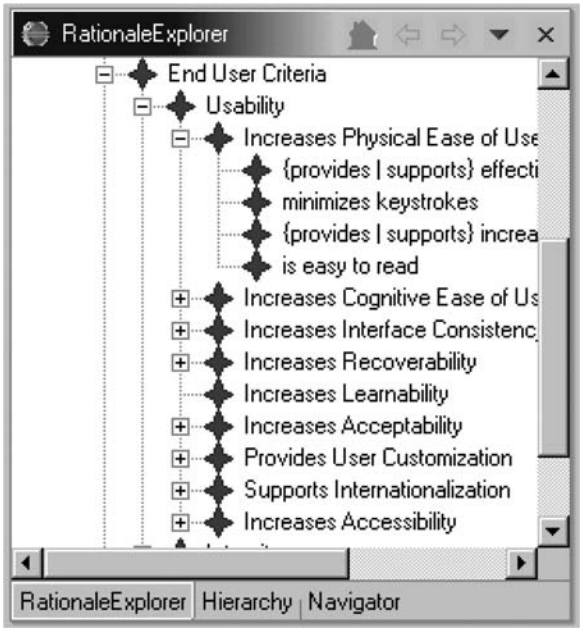

Fig. 13.4. Argument ontology for usability

Similar hierarchies have been developed for other high-level criteria in addition to Usability. One thing to note is that it is not a strict hierarchy – there are many cases where items contributing toward one criterion 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.

13.4.2 Capture

The goal behind the development of SEURAT was to evaluate potential *uses* of rationale. While this shifted the focus away from capture, there still needed to be a way to capture rationale using SEURAT in order for it to become a complete system. SEURAT facilitates this by being tightly integrated with the IDE being used to write the code. The developer is more likely to be willing to record their rationale if they do not need to start an additional tool to do so.

Editing screens were developed for each of the different rationale items supported by SEURAT and are accessible from the Rationale Explorer. Each item is created by selecting a context-sensitive menu item from its parent. Capture is also supported by automatic checking for rationale completeness. If the developer does not enter all the required rationale for a decision there will be an error indicated both in the Rationale Explorer and in the Rationale Task List.

13.4.3 Presentation

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 [15]. 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 13.5 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 that indicates a Java file. The associations are made by first selecting the Java element in the Package Explorer with the mouse, then selecting the alternative it implements in the Rationale Explorer, and then using a context-sensitive menu from the Rationale Explorer to indicate that the code and alternative are associated.

13.4.4 Inferencing

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. This support for inferencing classifies SEURAT as a prescriptive, as well as descriptive [12], rationale system.

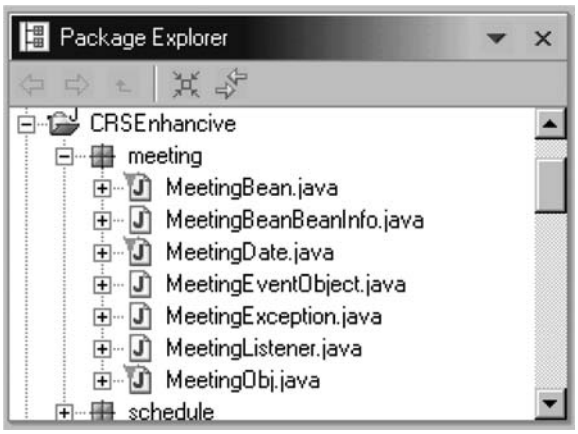

Fig. 13.5. Package explorer showing rationale associations

SEURAT supports four categories of inference: 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. These inferences point out cases where less-supported decisions were made by evaluating each alternative based on the number and importance of the arguments for and against it. These are not logical inferences but calculations of the relative value of the alternatives. 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.

The following sections give a few examples of some of the SEURAT inferences. Each inference can have many uses but, for convenience, we have grouped them by the type of maintenance being performed.

Corrective Maintenance

As mentioned earlier, a common source of error in software is when an assumption that was true when the system was developed no longer holds. SEURAT provides the ability to capture these assumptions during development. When the assumption is no longer valid, the maintainer can disable the assumption. SEURAT then performs inferencing over all portions of the rationale that refer to the assumption and re-evaluates the affected alternatives. If the removal of the assumption means that there are

selected alternatives that are no longer the best choice for their decision, the user will be informed of this.

One of the decisions that had to be made for the conference room scheduler system was how to specify the location of the room. There were two alternatives considered: combining the room and building names into a single string or specifying them separately. Figure 13.6 shows the Rationale Explorer after the assumption "customer normally combines room and building" has been disabled.

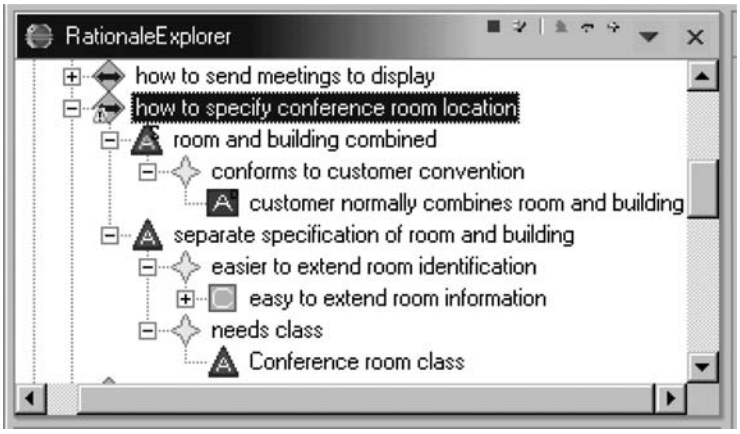

Fig. 13.6. Rationale explorer with disabled assumption

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 which combines them into one string (denoted by an "S" in the upper right-hand corner) is no longer the best supported (shown by the triangle icon with an exclamation point shown in the lower left-hand corner of the diamond shaped decision icon). The new warning is added to the bottom of the Rationale Task List shown in Fig. 13.7.

SEURAT also assists with corrective maintenance by providing access to any alternatives that have been either considered or implemented previously. This is useful both for pointing out what some possible corrections might be and to help make sure that a solution is not tried that was considered earlier and rejected. SEURAT also keeps track of dependencies between alternatives so that the user will be informed if they de-select an alternative on which another selected alternative depends.

For example, one response to the warning generated by the disabled assumption presented above would be to choose the other alternative, which separates out the specifications. This interacts with the alternative

selected for the decision about how to represent the conference room. If the room and building need to be displayed separately then they need to be stored separately in a conference room class, not combined as a string. Since the string representation alternative is currently selected, choosing an alternative that depends on there being a class for the conference room will give an error. Figure 13.8 shows this error as indicated on the Rationale Explorer (by the square with a white "X" in the middle appearing at the lower left-hand corner of the diamond-shaped decision icon) while Fig. 13.9 shows the error in the Rationale Task List. The entire explanation is available in SEURAT by either scrolling or resizing the Rationale Task List window of the SEURAT display.

Fig. 13.7. Rationale task list with new warning

Adaptive Maintenance

SEURAT supports adaptive maintenance by providing an easy way to evaluate the impact of any of the arguments in the Argument Ontology on the design and implementation. This is done by allowing the maintainer to perform "what-if" inferencing to see what might happen if their design priorities change. In SEURAT, each claim or argument can inherit its importance from importance values that the developer stored in the Argument Ontology. Lowering the importance of an argument will point out decisions that should probably change if that argument is no longer an important design goal. Increasing the importance of an argument will point out decisions that need to change if the argument becomes a higher priority.

One argument that was used very frequently in the conference room scheduling system was the argument that choosing an alternative would reduce development time because it was easy to code. Changing the importance of that argument showed places in the system where there may have been better alternatives that were not chosen because they were perceived to be more difficult. One example was for a decision of how to display error messages. Figure 13.10 shows the Rationale Explorer with the rationale for that decision. The importance of "Reduces Development

Time" has been decreased and the decision now has a warning (indicated by a triangle icon with an exclamation point in it on the lower left-hand corner of the decision icon) because the alternative of displaying errors as a line of text on the main display is now not supported as well as the alternative to display them in a pop-up box. Figure 13.11 shows the warning displayed on the Rationale Task List.

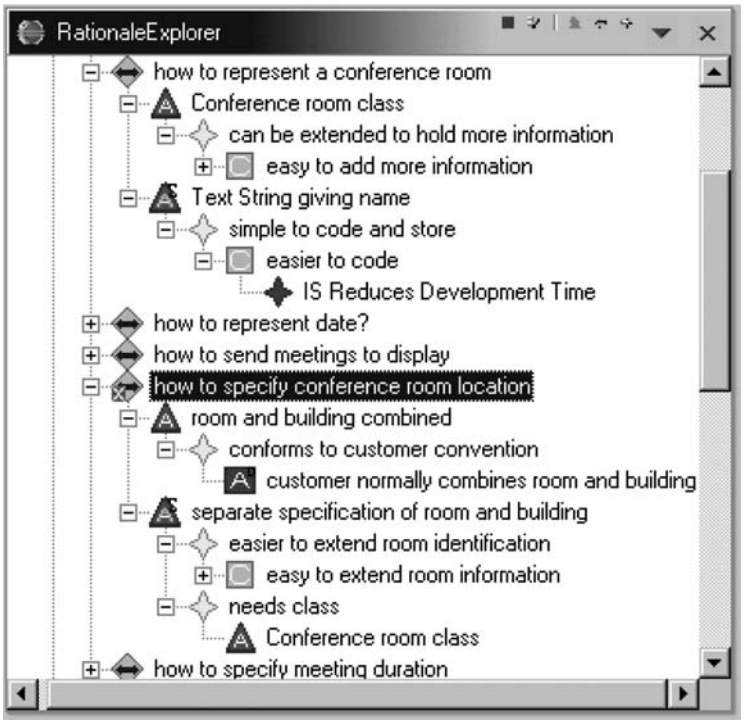

Fig. 13.8. Rationale explorer with decision error

Fig. 13.9. Rationale task list explaining the error

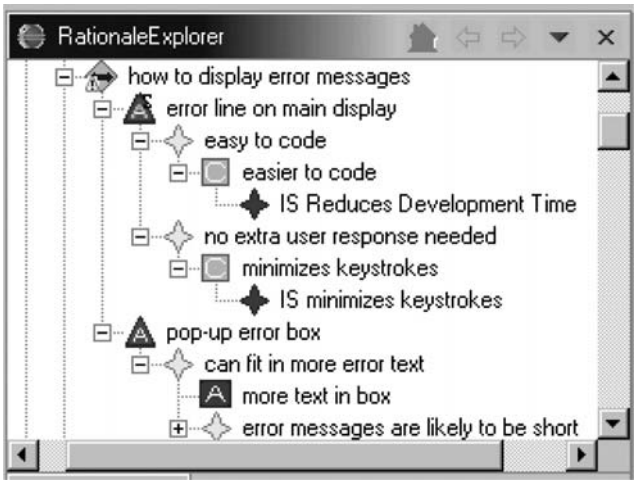

Fig. 13.10. Rationale explorer showing error message alternatives

Fig. 13.11. Rationale task list with the warning

Enhancive Maintenance

The inferences mentioned earlier as supporting other maintenance types will also support enhancive maintenance. During enhancive maintenance it is important to ensure that the rationale for decisions made when extending functionality are consistent with the rationale for the initial version(s) of the system. One way to do this is to make use of the tradeoffs background knowledge stored in SEURAT. Tradeoffs are used to indicate that there are two characteristics of the software that oppose each other and should always appear on opposite sides of an argument. The elements in the trade-off are both items from the Argument Ontology described earlier. The new decisions made during enhancive maintenance should consider both sides of the tradeoff and be consistent with the designer's original intent.

An example of this in the meeting scheduler is the tradeoff between increased flexibility and reduced development time. The developer has

added a tradeoff to SEURAT that indicates that if flexibility is increased, the amount of time to develop the system also increases. This is a non-symmetric tradeoff since increased development cost does not necessarily mean more flexibility. When the developer decided how to represent dates in the scheduling system, they chose to create a customized class to do this rather than using the Java Calendar class. This decision was made because the specialized class was thought to be more flexible. The cost of the new class was not considered. SEURAT detects that this is a tradeoff violation and warns the user. This lets the developer (or maintainer) know that the reasoning might not be complete. Figure 13.12 shows the rationale in the Rationale Explorer with the decision marked as having a problem (shown by the small triangle containing an exclamation point on the lower left-hand corner of the decision icon) and Fig. 13.13 shows the tradeoff explanation in the Rationale Task List. Note that the full explanation is available in SEURAT by scrolling across the window.

Another way that SEURAT can assist in checking decisions for consistency is by allowing arguments to inherit their importance from the global defaults stored in the Argument Ontology. If new decisions are made without overriding the defaults, SEURAT will evaluate them based on the same priorities as the rest of the design. This allows the software developer to define their priorities for the different nonfunctional requirements at a global level. This information will then propagate through the rationale when the different alternatives for a decision are evaluated and compared by SEURAT. If the best-evaluated alternative is not selected, the user will be informed both by warning icons in the Rationale Explorer (shown in Fig. 13.12) and warning descriptions in the Rationale Task List (shown in Fig. 13.13).

Fig. 13.12. Rationale for date representation

Fig. 13.13. Rationale task list with tradeoff violation

13.4.4 Rationale Retrieval

Both presentation of the rationale and inference over the rationale require that the system support efficient retrieval of the rationale elements. This is supported by storing the rationale in a MySQL database. The power of the relational database makes it possible for SEURAT to perform a number of different queries over the rationale. Several rationale queries, briefly mentioned in Sect. "Enhancive Maintenance", have been implemented in SEURAT. These include searching for entities of a particular type (requirement, decision, etc.); searching for requirements with a particular status (such as violated); searching for status messages that were overridden by the user so they could be re-enabled if necessary; and searching for claims and arguments where the default importance was overridden.

One interesting feature of SEURAT is the ability to look for common arguments occurring in the rationale. This can be valuable to the maintainers by giving them an overview of what the original developers thought was the most important criteria. The information can be shown for all alternatives or only for the selected ones.

13.5 SEURAT Evaluation

An initial evaluation was performed using SEURAT to assist with the three types of maintenance tasks described earlier: adaptive, corrective, and enhancive. Twenty subjects, a mixture of graduate students and industry professionals, were separated into control and experimental groups. The groups were divided in order to be balanced, based on their work experience and Java expertise. None of the subjects had used SEURAT before although some had attended research presentations describing the system. All were given a brief tutorial on how to use the system. The

control group used the Eclipse IDE alone to perform the tasks while the experimental group used Eclipse with the SEURAT plug-in and rationale that had been recorded for the system. The goal was to compare subject performance with and without access to rationale and the support of SEURAT. In this case, the primary performance measure was the time required to complete the task, not the quality of the result. This is because the tasks were relatively simple in order to allow the experiment to be completed in a reasonable amount of time (less than 4 h per subject). The system being modified was the Conference Room Scheduling System described earlier. This was a Java program that had been originally written five years earlier as a meeting scheduler and had been adapted over the years to schedule meetings in multiple rooms. It had many characteristics of legacy code, such as using an obsolete version of Java and having been written by multiple developers.

Each subject was timed for each task with two times being measured: the time required to find the portion of the code that needed to be changed to complete the task and the time required to complete the task. The results were not statistically significant, suggesting that more experiments need to be performed, but the group using SEURAT did perform better on average than the control group. In addition, SEURAT helped nonexperts more than experts. We would expect this result to change when more challenging maintenance tasks are used. Figure 13.14 shows the average times for the delta (time to find change) and total time for each task.

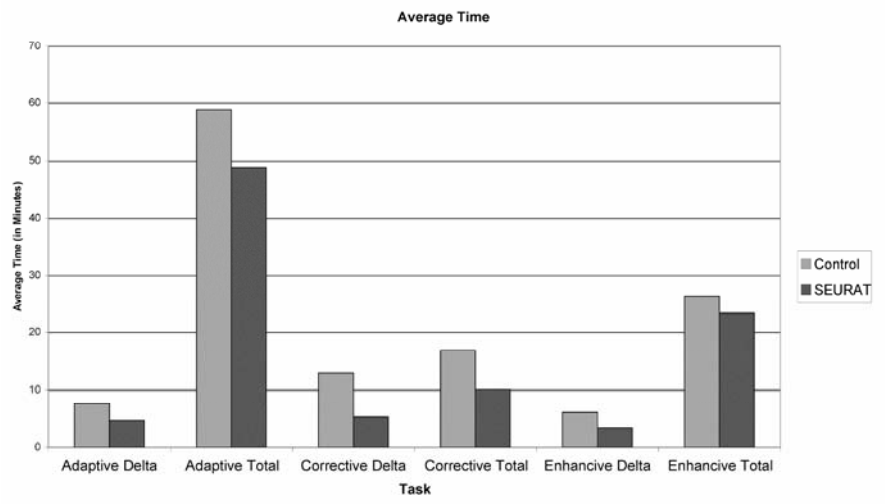

Fig. 13.14. Average times for each task

A survey asking the subjects who used SEURAT what they thought of it was also administered. These questions asked the subjects to give their opinion on a Likert scale where SA means Strongly Agree, A means Agree, U means Undecided, D means Disagree, and SD means Strongly Disagree. Figure 13.15 shows the summary of these results. The survey results indicated that the majority of the participants using SEURAT thought it was a useful tool and that it assisted them in performing the tasks given in the experiment.

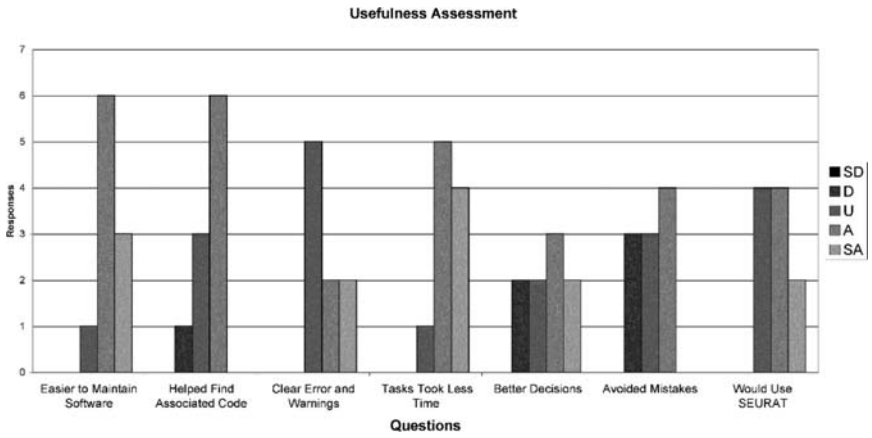

Fig. 13.15. SEURAT usefulness survey results

13.6 Conclusions and Recommendations

A way to help reduce the risk, and thereby the cost, of software maintenance is to give the maintainers insight into the intent behind the original design, i.e., the design rationale. This can be made even more useful if new changes can be checked to ensure that the reasoning behind new decisions is consistent with the original system. Conversely, if the goals of the system have changed, it would be useful if the maintainer could know how that would affect decisions made earlier.

To support DR use, we have developed the SEURAT system, which tightly integrates with an IDE to support the entry of, display of, and inferencing over the rationale. SEURAT allows the maintainer to take advantage of the knowledge captured during initial development to assist in maintenance changes, both by helping the maintainer figure out what needs to be changed and by verifying that new additions are consistent

with the designers original intent. This is helpful for all types of software maintenance.

One of the next steps planned for SEURAT is integration with additional tools used at different stages of the design process. These would include requirements tools, design tools, and possibly testing tools. This would allow us to continue to investigate the differences in the rationale generated and used at different stages in the development process. The goal is for SEURAT to be used during all stages of development by augmenting current development process and practice to support rationale capture and use. We also will address scalability concerns to transition SEURAT from a research prototype to a tool that can be used in full-scale software development.

We would also like to extend SEURAT to handle multiuser rationale. One thing that rationale can be very useful for is to capture the different viewpoints expressed by team members while making decisions [23]. It would be interesting to explore how capturing potentially conflicting information from different developers could be used in evaluating the design decisions. We also want to investigate systems that SEURAT could be interfaced with to assist in the capture process. Some possible sources of rationale are configuration management and problem reporting systems.

We feel that a system like SEURAT would be invaluable during software maintenance. The SEURAT system contributes a detailed, reusable list of reasons for making software decisions in the Argument Ontology. SEURAT then uses those reasons to support semantic inferencing to determine the impact of these decisions on the software system (and to promote consistency in the rationale). SEURAT also provides an integrated environment where rationale capture and use can be performed using the same tools that are used in development and maintenance. There are many benefits to having the design rationale available during maintenance but only with appropriate system support, such as that provided by SEURAT, can rationale live up to its full potential.

References

[1] Boehm B, Bose P (1994) A collaborative spiral software process model based on theory W. In: Proceedings of the International Conference on the Software Process, Reston, VA, pp. 59–68

[2] Bose P (1995) A model for decision maintenance in the WinWin Collaboration Framework. In: Proceedings of the Conference on Knowledge-based Software Engineering, Boston, MA, pp. 105–113

[3] Bratthall L, Johansson E, Regnell B (2000) Is a design rationale vital when predicting change impact? A controlled experiment on software architecture

evolution. In: Proceedings of the International Conference on Product Focused Software Process Improvement, Oulu, Finland, pp. 126–139
[4] Burge JE (2005) Software Engineering Using design RATionale. Ph.D. thesis, Worcester Polytechnic Institute
[5] Burge JE, Brown DC (2002) NFRs: fact or fiction? Technical Report WPI-CS-TR-02-01. Worcester Polytechnic Institute.
[6] Burge JE, Brown DC (2004) An integrated approach for software design checking using rationale. In: Design Computing and Cognition '04, Gero J (ed). Kluwer Academic, Dordrecht, pp. 557–576
[7] Canfora G, Casazza G, De Lucia A (2000) A Design rationale based environment for cooperative maintenance. International Journal of Software Engineering and Knowledge Engineering 10(5):627–645
[8] Chapin N (2000) Software maintenance types – a fresh view. In: Proceedings of the International Conference on Software Maintenance, San Jose, CA, pp. 247–252
[9] Conklin J, Burgess-Yakemovic K (1995) A process-oriented approach to design rationale. In: Design Rationale Concepts, Techniques, and Use, Moran T, Carroll J (eds.). Lawrence Erlbaum, Mahawah, NJ, pp. 293–428
[10] Chung L, Nixon BA, Yu E, Mylopoulos J (2000) Non-Functional Requirements in Software Engineering. Kluwer Academic, Dordrecht,
[11] Dutoit AH, Paech B (2001) Rationale management in software engineering. In: Handbook of Software Engineering and Knowledge Engineering, Chang SK (ed). World Scientific, Singapore, pp. 787–816
[12] Dutoit AH, McCall R, Mistrik I, Paech B (2006) Rationale management in software engineering: Concepts and Techniques. In: Rationale Management in Software Engineering, Dutoit AH, McCall R, Mistrik I, Paech B (eds.). Springer, Berlin Heidelberg New York, pp. 1–48
[13] Filman RE (1998) Achieving ilities. In: Proceedings of the Workshop on Compositional Software Architectures, Monterey CA
[14] Fischer G, Lemke A, McCall R, Morch A (1995) Making argumentation serve design. In: Design Rationale Concepts, Techniques, and Use, Moran T, Carroll J (eds.). Lawrence Erlbaum, Mahawah, NJ, pp. 267–294
[15] Karsenty L (1996) An empirical evaluation of design rationale documents. In: Proceedings of the Conference on Human Factors in Computing Systems. Vancouver, BC, pp. 150–156
[16] Klein M (1997) An exception handling approach to enhancing consistency, completeness and correctness in collaborative requirements capture. Concurrent Engineering Research and Applications 5(1):37–46
[17] Lee J (1991) Extending the Potts and Bruns model for recording design rationale. In: Proceedings of the International Conference on Software Engineering, Austin, TX, pp. 114–125
[18] Lee J (1997) Design rationale systems: understanding the issues. IEEE Expert 12(3): 78–85
[19] Lehman M (2003) Software evolution cause or effect? Stevens Award Lecture: International Conference on Software Maintenance, Amsterdam

[20] Lientz BP, Swanson EB (1988) Software Maintenance Management. Addison-Wesley, Reading, MA
[21] Lougher R, Rodden T (1993) Group support for the recording and sharing of maintenance rationale. Software Engineering Journal 8(6):295–306
[22] MacLean A, Young RM, Bellotti V, Moran TP (1995) Questions, Options and Criteria: Elements of Design Space Analysis In: Design Rationale Concepts, Techniques, and Use, Moran T, Carroll J (eds.). Lawrence Erlbaum, Mahawah NJ, pp. 201–251
[23] Peña-Mora F, Sriram D, Logcher R (1995) Design rationale for computer-supported conflict mitigation. ASCE Journal of Computing in Civil Engineering 9(1):57–72
[24] Peña-Mora F, Vadhavkar S (1996) Augmenting design patterns with design rationale. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 11(2):93–108
[25] Potts C, Bruns G (1988) Recording the reasons for design decisions. In: Proceedings of the International Conference on Software Engineering. Singapore, pp. 418–427
[26] Reiss SP (2002) Constraining software evolution. In: Proceedings of the International Conference on Software Maintenance, Montreal, Que. Canada, pp. 162–171