

Worcester Polytechnic Institute Digital WPI

Doctoral Dissertations (All Dissertations, All Years)

Electronic Theses and Dissertations

2005-05-02

Software Engineering Using design RATionale

Janet E. Burge

Worcester Polytechnic Institute

Follow this and additional works at: <https://digitalcommons.wpi.edu/etd-dissertations>

Repository Citation

Burge, J. E. (2005). *Software Engineering Using design RATionale*. Retrieved from <https://digitalcommons.wpi.edu/etd-dissertations/244>

This dissertation is brought to you for free and open access by [Digital WPI](#). It has been accepted for inclusion in Doctoral Dissertations (All Dissertations, All Years) by an authorized administrator of Digital WPI. For more information, please contact wpi-etd@wpi.edu.

Software Engineering Using design RATIONale

by

Janet E. Burge

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

May 2005

APPROVED:

Dr. David C. Brown, Major Advisor

Dr. George T. Heineman, Committee Member

Dr. Feniosky Peña-Mora, External Committee Member
University of Illinois Urbana-Champaign

Dr. Carolina Ruiz, Committee Member

Dr. Michael Gennert, Head of Department

Abstract

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 is invaluable as an aid for revising, maintaining, documenting, evaluating, and learning the design.

The presence of DR would be especially valuable for software maintenance. The rationale would provide insight into why the system is the way it is by giving the reasons behind the design decisions, could help to indicate where changes might be needed during maintenance if design goals change, and help the maintainer avoid repeating earlier mistakes by explicitly documenting alternatives that were tried earlier that did not work.

Unfortunately, while everyone agrees that design rationale is useful, it is still not used enough in practice. Possible 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 few systems available to support rationale use and capture. We have addressed this problem by developing and evaluating a system called SEURAT (Software Engineering Using RATIONale) which integrates with a software development environment and goes beyond mere presentation of rationale by inferencing over it to check for completeness and consistency in the reasoning used while a software system is being developed and maintained.

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.

Acknowledgements

This dissertation could never have been completed without the support of many people over the past eight years.

The advice and support of my advisor, David C. Brown, has been immeasurable. His knowledge of the area was of considerable help, of course, as was his experience as a researcher, but I appreciated even more the amount of time he was willing to dedicate toward my growth as a researcher. He knew the right time to put on pressure to keep me moving in the right direction but also when to give encouragement and space when things got rough. I would never have made it through these years without him and cannot imagine having a better advisor.

Many in the WPI CS department have supported me during this time. My Ph.D. committee provided considerable guidance on my research. I was able to present and discuss this work in two research groups: the AI Research Group (AIRG) that Carolina Ruiz coordinated for much of my time at WPI as well as the AI in Design Research Group (AIDG) lead by David Brown. George Heineman let me participate in his Software Engineering Research Group (SERG) and also gave me invaluable support in preparing to teach my graduate software engineering course. Karen Lemone first offered me a TA position and gave me the incentive to leave my job to go to school full time. Steve Taylor, Stanley Selkow, Isabel Cruz, and Glynis Hamel put up with my fledgling TA efforts. Glynis Hamel then helped me put together my own undergraduate class with considerable help from her extensive notes and files. I would never have survived that experience without her help. Micha Hofri let me teach my first undergraduate class and Mike Gennert let me teach my graduate course. I'm very grateful to both of them for giving me the experience and especially to Mike for his encouragement to keep trying even though my first teaching experiences were somewhat discouraging. Mike and Dave were both very generous with funding over the years both for conference travel and for funding the experiment described in this dissertation. There are very few faculty members in the WPI CS department who did not get involved in my education here either in teaching a course I took or in giving me advice on my research or teaching. I also appreciate the help and support I received from my fellow graduate students, especially those who were willing to

serve as experiment subjects. I greatly appreciate the time they gave to contribute towards this dissertation.

I'd like to give special thanks to one professor who is no longer with us: Dr. Lee Becker. Lee's optimism and encouragement is something that I'll always remember when faced with adversity. When I started to have doubts and did not think it would be possible to finish I thought of what Lee would say and just kept going.

I also was very fortunate to get financial support through a very flexible part-time job at Charles River Analytics. Eva Hudlicka, who also co-advised my MS thesis, helped me get the job. I am very grateful to Greg Zacharias, president, and my immediate supervisor, Paul Gonsalves, for their infinite patience with my constantly changing work hours. I also appreciate the support I received from my many colleagues and friends there. There were many times when someone had to pick up something that I needed to drop because of my academic obligations yet this was done with grace and understanding. Several of my co-workers also participated in the SEURAT evaluation experiment and to their credit no one who I asked to participate (in person) turned me down.

I have many friends who supported me during this time. My friend Shirley Rieven inspired me to go back to school through her example. She and her husband Steve have been a great support to me over the twenty some years that I have known them. My friend Mark Mezger was always there to listen to me complain and go catch a movie. Mikhail Mikhailov watched countless SEURAT presentations, gave InfoRat its name, and helped me create embedded postscript diagrams when I was forced to submit documents in LaTeX. The D&D group (Eric, Amy, Anne, Kevin, and Steve) gave me a much needed escape from reality once a week. Chuck Jones and Pam Barz were there to celebrate special occasions with me. I also escaped my studies during game night with Thom and Colleen Goodsell and the rest of the game-night gang (Rachel, Alan, Igor, Marina, Sean, Darlene, Anne, Kurt, Magnus, Adine, and more) and my with my friends in Symphony Pro Musica. I'm also blessed with wonderful neighbors who did not complain when my lack of funds turned my house into the neighborhood eyesore. Their understanding and tolerance was truly outstanding.

Of course, my family offered considerable support during this time. Not too many parents would be so encouraging when their 30-something child announces they are quitting their job to go back to school. Of course, it is all their fault since my father had been talking for years about how being a college professor was the ideal job. This is not a job he has held so I'm not sure why I believed him. I hope he's right!

Table of Contents

CHAPTER 1	Introduction.....	1
1.1.	The Goal of the Research.....	2
1.2.	The Importance of the Research	2
1.3.	Expected Benefits	2
1.4.	Design Rationale	3
1.5.	Why Isn't Rationale Used Now?.....	4
1.6.	The Challenge of DR in Software Engineering	6
1.7.	Expected Results	6
1.8.	Organization of the Dissertation	8
CHAPTER 2	The Problem.....	9
2.1.	Design Rationale and the Software Development Process	9
2.2.	Uses of Rationale in Software Development and Maintenance....	12
2.3.	Encouraging DR Use in Software Design.....	14
2.4.	Summary	15
CHAPTER 3	Relevant Research.....	16
3.1	Design	16
3.2	AI in Design	17
3.3	Design Rationale	18
3.3.1	Design Rationale Representation	18
3.3.2	Design Rationale Capture	20
3.3.3	Design Rationale Use	22

3.3.4	Design Rationale for Software Design	23
3.4	Software Design	25
3.4.1	Software Development Processes	25
3.4.2	Requirements Engineering	26
3.4.3	Software Architecture	28
3.4.4	Software Maintenance	29
3.5	Summary	30
CHAPTER 4	Investigating DR Uses: Inferencing over Design Rationale	32
4.1	Using Rationale for Validation and Evaluation.....	32
4.2	Prototype System for Inferencing Over Rationale.....	32
4.2.1	Approach	32
4.2.2	Inferences	37
4.2.3	Vocabulary	39
4.2.4	Tradeoff Evaluation	40
4.3	Implementation and Examples.....	42
4.3.1	Browse Rationale	42
4.3.2	Modify Rationale	45
4.4	Summary	46
CHAPTER 5	The Software Development Process and Rationale	48
5.1	Study Goals	48
5.2	Study Description.....	49
5.2.1	Initial Design	49
5.2.2	Corrective Maintenance - Minor Bug in the Program	51
5.2.3	Adaptive Maintenance - Revisiting the Design for Usability ..	51
5.2.4	Enhancive Maintenance - Extending the Functionality	51
5.3	Study Results.....	52
5.3.1	Initial Design	52
5.3.2	Corrective Maintenance	54
5.3.3	Adaptive Maintenance	54
5.3.4	Enhancive Maintenance	55

5.4	Summary and Conclusions.....	55
CHAPTER 6	The Approach	57
6.1.	Uses of Rationale for Software Maintenance	58
6.2.	Tool Support for Rationale Use.....	60
CHAPTER 7	Software Engineering Using RAtionale (SEURAT)	61
7.1.	System Architecture	61
7.2.	Rationale Representation	62
7.2.1.	Motivation	63
7.2.2.	Related Work	64
7.2.3.	Representation Format	68
7.3.	Inferences Supported.....	90
7.3.1.	Syntactic	91
7.3.2.	Semantic	92
7.3.3.	Queries	95
7.3.4.	Historical	96
7.4.	Argument Ontology	96
7.4.1.	Affordability Criteria	98
7.4.2.	Adaptability Criteria	99
7.4.3.	Dependability Criteria	101
7.4.4.	End User Criteria	102
7.4.5.	Needs Satisfaction Criteria	104
7.4.6.	Maintainability Criteria	104
7.4.7.	Performance Criteria	104
7.5.	Rationale Entry and Presentation.....	105
CHAPTER 8	SEURAT Software Design and Implementation	107
8.1.	SEURAT Software Architecture	107
8.2.	Rationale Repository and Argument Ontology.....	111
8.2.1.	Requirements	111
8.2.2.	Decision	112
8.2.3.	Alternative	113

8.2.4.	Argument	114
8.2.5.	Claim	116
8.2.6.	Assumption	116
8.2.7.	Ontology Entry	117
8.2.8.	Tradeoffs	118
8.2.9.	Questions	119
8.2.10.	History	119
8.2.11.	Status	120
8.2.12.	Associations	121
8.3.	Rationale Explorer	122
8.3.1.	Requirement Menu	125
8.3.2.	Decision Menu	125
8.3.3.	Alternative Menu	126
8.3.4.	Argument Menu	126
8.3.5.	Claim Menu	126
8.3.6.	Assumption Menu	127
8.3.7.	Question Menu	127
8.3.8.	Tradeoff and Co-Occurrence Menu	127
8.3.9.	Ontology Entry Menu	127
8.4.	Inference Engine	128
8.4.1.	Error and Warning Visualization	128
8.4.2.	Error and Warning Detection	130
8.5.	Rationale to Code Associations	134
8.6.	Rationale Display and Editing	137
8.6.1.	Requirement	137
8.6.2.	Decision	138
8.6.3.	Alternative	138
8.6.4.	Argument	139
8.6.5.	Claim	141
8.6.6.	Assumption	141
8.6.7.	Question	141
8.6.8.	Tradeoff	142
8.6.9.	Co-occurrence	143

8.6.10.	Ontology Entry	143
8.7.	Rationale Query Interface	144
8.7.1.	Find Rationale Entity	144
8.7.2.	Find Common Arguments	145
8.7.3.	Find Requirements	146
8.7.4.	Find Status Overrides	147
8.7.5.	Find Importance Overrides	148
CHAPTER 9	System Demonstration	149
9.1.	Software Maintenance Examples	149
9.1.1.	Adaptive Maintenance	149
9.1.2.	Corrective Maintenance	155
9.1.3.	Enhancive Maintenance	157
9.2.	Inferencing Examples	160
9.2.1.	Changing Priorities	160
9.2.2.	Disabling Assumptions	162
9.3.	Summary	165
CHAPTER 10	Evaluation	166
10.1.	Experiment Design	166
10.1.1.	Experiment Goals	166
10.1.2.	Experiment Design	166
10.1.3.	Experiment Subject Selection	172
10.2.	Experiment Results	173
10.2.1.	Support for Maintenance	173
10.2.2.	SEURAT Usability	188
10.2.3.	SEURAT Usefulness	189
10.2.4.	Experiment Evaluation	191
10.3.1.	Experiment Shortcomings	192
10.3.2.	Suggested Improvements/Additional Experiments	194
CHAPTER 11	Conclusion	196
11.1.	Contributions	197

11.2.	Future Work	198
11.2.1.	Investigation of Rationale for Different Phases	199
11.2.2.	Multi-User Rationale	199
11.2.3.	Longer-Term SEURAT Study	200
11.2.4.	Rationale Capture	200
11.3.	Summary	200
	References.....	202
APPENDIX A	SEURAT User's Guide.....	A-1
A.1.	What is SEURAT?	A-5
A.2.	Rationale in SEURAT.....	A-6
A.2.1.	Rationale Structure	A-6
A.2.2.	Entering New Rationale	A-10
A.2.3.	Editing Existing Rationale	A-17
A.3.	The Rationale-Code Connection.....	A-17
A.3.1.	Associating Rationale with Code	A-17
A.3.2.	Finding Associated Rationale	A-18
A.3.3.	Removing Rationale Associations	A-19
A.4.	Rationale Tasks	A-20
A.5.	Rationale Queries	A-21
A.5.1.	Find Rationale Entity	A-21
A.5.2.	Find Common Arguments	A-22
A.5.3.	Find Requirements	A-23
A.5.4.	Find Status Overrides	A-24
A.5.5.	Find Importance Overrides	A-25
A.6.	Using the Rationale.....	A-25
A.6.1.	Modifying Importance Values	A-26
A.6.2.	Disabling Rationale Items	A-26
APPENDIX B	Experimental Surveys	B-1

List of Figures

FIGURE 2-1.	Software Development Phases and Rationale	11
FIGURE 3-1.	SEURAT Relationships.....	31
FIGURE 4-1.	Design Rationale in the Design Process	33
FIGURE 4-2.	Intersection Diagram.....	34
FIGURE 4-3.	Design Rationale Elements	35
FIGURE 4-4.	Subset of Alternatives for Requirement “Four Traffic Lights”	36
FIGURE 4-5.	Goals and Sub-goals for the Unsatisfied Requirement	37
FIGURE 4-6.	Unsatisfied Requirement Check	38
FIGURE 4-7.	Arguments Against Outweigh For	38
FIGURE 4-8.	Best Alternative Not Chosen	38
FIGURE 4-9.	Contradictory Arguments.....	39
FIGURE 4-10.	Completeness for a Goal.....	39
FIGURE 4-11.	Consistency for a Goal.....	39
FIGURE 4-12.	Standard Claim Vocabulary	40
FIGURE 4-13.	User Defined Claim Vocabulary	40
FIGURE 4-14.	Causality Violation	41
FIGURE 4-15.	Missing Claim.....	41
FIGURE 4-16.	Tradeoff Importance Violation.....	42
FIGURE 4-17.	Requirement Listing	42
FIGURE 4-18.	Goal Listing	43
FIGURE 4-19.	Alternative Listing	43
FIGURE 4-20.	Alternative Blinking Red/Yellow.....	44
FIGURE 4-21.	Version History	44
FIGURE 4-22.	Modify Rationale Options	45
FIGURE 4-23.	Modify Alternative Options.....	46
FIGURE 5-1.	Rationale Components	50
FIGURE 7-1.	SEURAT System Architecture.....	62
FIGURE 7-2.	Representation Elements.....	70
FIGURE 7-3.	RATSpeak Argumentation Structure	70
FIGURE 7-4.	Rationale Top Level Representation.....	71
FIGURE 7-5.	Requirement Schema	72

FIGURE 7-6.	Requirement Status	73
FIGURE 7-7.	History and History Records.....	73
FIGURE 7-8.	Requirement Example.....	73
FIGURE 7-9.	Decision Problem Schema	74
FIGURE 7-10.	Decision Type	75
FIGURE 7-11.	Development Phase.....	75
FIGURE 7-12.	Decision Status	76
FIGURE 7-13.	Example Decision Problem	76
FIGURE 7-14.	Question Schema	77
FIGURE 7-15.	Question Status Schema.....	78
FIGURE 7-16.	Question Example.....	78
FIGURE 7-17.	Alternative Schema.....	79
FIGURE 7-18.	Alternative Status Schema	79
FIGURE 7-19.	Alternative Example	80
FIGURE 7-20.	Argument Schema.....	81
FIGURE 7-21.	Argument Type Schema.....	82
FIGURE 7-22.	Importance Schema.....	82
FIGURE 7-23.	Amount Schema.....	83
FIGURE 7-24.	Plausibility Schema.....	83
FIGURE 7-25.	Argument Example	84
FIGURE 7-26.	Claim Schema	84
FIGURE 7-27.	Direction Schema.....	85
FIGURE 7-28.	Claim Example	85
FIGURE 7-29.	Assumption Schema	86
FIGURE 7-30.	Assumption Example.....	86
FIGURE 7-31.	Argument Ontology Schema.....	86
FIGURE 7-32.	Ontology Entry Schema.....	87
FIGURE 7-33.	Argument Ontology Example	88
FIGURE 7-34.	Background Knowledge Schema.....	88
FIGURE 7-35.	Tradeoff Schema	89
FIGURE 7-36.	Co-Occurrence Relationship Schema	89
FIGURE 7-37.	Background Knowledge Example	90
FIGURE 7-38.	Affordability Criteria	100
FIGURE 7-39.	Adaptability Criteria	101
FIGURE 7-40.	Dependability Criteria.....	102
FIGURE 7-41.	End User Criteria	103
FIGURE 7-42.	Needs Satisfaction Criteria	104
FIGURE 7-43.	Maintainability Criteria.....	104
FIGURE 7-44.	Performance Criteria.....	105
FIGURE 8-1.	SEURAT Software Architecture.....	108
FIGURE 8-2.	Rationale Update Flowchart	110

FIGURE 8-3.	SEURAT and Eclipse.....	111
FIGURE 8-4.	Requirement SQL Definition.....	112
FIGURE 8-5.	Decision SQL Definition	113
FIGURE 8-6.	Alternative SQL Definition	114
FIGURE 8-7.	Argument SQL Definition	115
FIGURE 8-8.	Claim SQL Definition.....	116
FIGURE 8-9.	Assumption SQL Definition	117
FIGURE 8-10.	Ontology Entry SQL Definition	118
FIGURE 8-11.	Ontology Relationships SQL Definition.....	118
FIGURE 8-12.	Tradeoff SQL Definition.....	119
FIGURE 8-13.	Question SQL Definition	120
FIGURE 8-14.	History SQL Definition	120
FIGURE 8-15.	Status SQL Definition.....	121
FIGURE 8-16.	Association SQL Definition.....	122
FIGURE 8-17.	Rationale Explorer – Top Level Rationale.....	122
FIGURE 8-18.	Rationale Explorer with Expanded Rationale.....	123
FIGURE 8-19.	Rationale Element Icons	124
FIGURE 8-20.	Icon Overlay Examples.....	124
FIGURE 8-21.	Requirement Relationship Display	125
FIGURE 8-22.	Rationale Task List	128
FIGURE 8-23.	Package Explorer with Associations.....	135
FIGURE 8-24.	Association Icon	135
FIGURE 8-25.	Bookmark View	136
FIGURE 8-26.	Alternative Showing Code Association.....	137
FIGURE 8-27.	Requirement Editor.....	138
FIGURE 8-28.	Decision Editor	139
FIGURE 8-29.	Alternative Editor	140
FIGURE 8-30.	Argument Editor	140
FIGURE 8-31.	Claim Editor.....	141
FIGURE 8-32.	Assumption Editor	142
FIGURE 8-33.	Question Editor	142
FIGURE 8-34.	Tradeoff Editor.....	143
FIGURE 8-35.	Ontology Entry Editor	143
FIGURE 8-36.	Find Entity Display	144
FIGURE 8-37.	Select Claim Display	145
FIGURE 8-38.	Find Common Arguments.....	145
FIGURE 8-39.	Common Argument Display	146
FIGURE 8-40.	Find Requirements Display	146
FIGURE 8-41.	Addressed Requirements	147
FIGURE 8-42.	Status Override Display	147
FIGURE 8-43.	Importance Override Display.....	148

FIGURE 9-1.	Rationale for “how to store user information”	150
FIGURE 9-2.	Warning Message for “save in a text file”	150
FIGURE 9-3.	Decision “how to store user information”	151
FIGURE 9-4.	Alternative “serialize user information”	152
FIGURE 9-5.	Alternative “save in a text file”	153
FIGURE 9-6.	Bookmark Showing Association for “save in a text file”	153
FIGURE 9-7.	Method Where User Information is Saved	154
FIGURE 9-8.	Rationale for “where to load user information”	154
FIGURE 9-9.	Rationale for “which week to display when room changes”	155
FIGURE 9-10.	Error for the Selected Alternative	156
FIGURE 9-11.	Bookmark Display Showing Association for “display meetings for current week”	156
FIGURE 9-12.	Method Where Dates are Reset	157
FIGURE 9-13.	Violated Requirement “Administrator can cancel any meeting”	158
FIGURE 9-14.	Error Showing Violated Requirement	158
FIGURE 9-15.	Rationale for “who is allowed to cancel meetings”	159
FIGURE 9-16.	Bookmark Showing Association for “only owner can cancel meetings”	159
FIGURE 9-17.	Method Where Meetings are Cancelled	160
FIGURE 9-18.	Common Argument Display with Ontology Entries	161
FIGURE 9-19.	Ontology Entry “Reduces Development Time”	161
FIGURE 9-20.	Rationale Task Display with New Warnings	162
FIGURE 9-21.	Assumptions for Conference Room Scheduling System	163
FIGURE 9-22.	Assumption “standard working hours 8 to 6”	163
FIGURE 9-23.	Rationale for Decision “schedule duration”	164
FIGURE 9-24.	Rationale Task List with New Warning	164
FIGURE 10-1.	Adaptive Maintenance - Time to Find Change	174
FIGURE 10-2.	Adaptive Maintenance - Time to Complete Task	174
FIGURE 10-3.	Adaptive Maintenance - Variance in Time to Find the Change	175
FIGURE 10-4.	Adaptive Maintenance - Variance in Time to Complete Task	176
FIGURE 10-5.	Adaptive Maintenance - Time to Find Change vs. Experience	176
FIGURE 10-6.	Adaptive Maintenance - Time to Complete Task vs. Experience	177
FIGURE 10-7.	Corrective Maintenance - Time to Find Change	178
FIGURE 10-8.	Corrective Maintenance - Time to Complete Task	178
FIGURE 10-9.	Corrective Maintenance - Variance in Time to Find Change	179
FIGURE 10-10.	Corrective Maintenance - Variance in Time to Complete Task	180
FIGURE 10-11.	Corrective Maintenance - Time to Find Change vs. Experience	180
FIGURE 10-12.	Corrective Maintenance - Time to Complete Task vs. Experience	181
FIGURE 10-13.	Enhancive Maintenance - Time to Find Change	182
FIGURE 10-14.	Enhancive Maintenance - Time to Complete Task	182
FIGURE 10-15.	Enhancive Maintenance - Variance in Time to Find Change	183
FIGURE 10-16.	Enhancive Maintenance -- Variance in Time to Complete Task	184
FIGURE 10-17.	Enhancive Maintenance - Time to Find Change vs. Experience	184

FIGURE 10-18.	Enhancive Maintenance -- Time to Complete Task vs. Experience	185
FIGURE 10-19.	Usefulness Summary	189
FIGURE A-1.	SEURAT Main Display	A-6
FIGURE A-2.	Relationships between rationale entities.....	A-8
FIGURE A-3.	Rationale Explorer	A-9
FIGURE A-4.	Rationale Icons	A-10
FIGURE A-5.	Requirement Editor.....	A-11
FIGURE A-6.	Decision Editor	A-12
FIGURE A-7.	Alternative Editor	A-13
FIGURE A-8.	Argument Editor	A-14
FIGURE A-9.	Claim Editor.....	A-14
FIGURE A-10.	Assumption Editor	A-15
FIGURE A-11.	Question Editor	A-15
FIGURE A-12.	Tradeoff Editor.....	A-16
FIGURE A-13.	Ontology Entry Editor	A-17
FIGURE A-14.	Package Explorer with Associations.....	A-18
FIGURE A-15.	Bookmark View	A-18
FIGURE A-16.	Alternative Showing Code Association.....	A-19
FIGURE A-17.	Rationale Task List	A-20
FIGURE A-18.	Find Entity Display	A-21
FIGURE A-19.	Select Claim Display	A-22
FIGURE A-20.	Find Common Arguments.....	A-22
FIGURE A-21.	Common Argument Display	A-23
FIGURE A-22.	Find Requirements Display	A-23
FIGURE A-23.	Addressed Requirements	A-24
FIGURE A-24.	Status Override Display	A-24
FIGURE A-25.	Importance Override Display.....	A-25

List of Tables

TABLE 7-1.	DRL/RATSpeak Comparison	65
TABLE 8-1.	Rationale Task Messages	129
TABLE 10-2.	GQM Analysis Results	167
TABLE 10-3.	Experimental Group Summary	173
TABLE 10-4.	F-Test Analysis Results.....	186
TABLE 10-5.	ANACOVA Analysis Results	186
TABLE 10-6.	Mann-Whitney Analysis Results	187
TABLE 10-7.	GQM Analysis with Results	193

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.

The presence of DR would be especially valuable for software development. This is true for several reasons. One is that the inherent mutability of software increases the chance that it will be modified during its lifetime. Another is that the software lifetimes tend to be long, often longer than expected, and this requires that it be changed as the world around it does. Software maintenance is a very expensive part of the software development process and is made more difficult because the original designers are often not available. This is an area where we feel that DR could be of some assistance. The rationale would provide insight into why the system is the way it is by giving the reasons behind the design decisions, could help to indicate where changes might be needed during maintenance if design goals change, and help the maintainer avoid repeating earlier mistakes by explicitly documenting alternatives that were tried earlier that did not work.

In this chapter we discuss the goal of the research (Section 1.1), the potential importance of the research (Section 1.2), the expected benefits of the research (Section 1.3), design rationale (1.4), why rationale is not used now (1.5), the challenge of using design rationale in software engineering (1.6), methods and expected results for the research (1.7), and an outline of this dissertation (1.8).

1.1. The Goal of the Research

Design Rationale has significant potential for having great value to the software developer, yet it is rarely captured in practice. One reason for this is that it is unclear whether the cost of collecting this information is outweighed by its usefulness. In this research, we chose to investigate how DR can be used during software maintenance and developed a system that would support those uses. The goal, in summary, is to show that with appropriate tool support, rationale can provide useful support to the software maintainer. This is an important step in the direction toward showing that the cost of rationale capture can be justified during software development.

1.2. The Importance of the Research

Software development is an interesting application for DR in a number of ways. In one sense, software development is really system development where the code itself 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 meet the changing needs of the user, and as it is adapted to respond to changes in the underlying technology.

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. This would fit nicely into a software development environment such as the one proposed by Nuseibeh, et. al. [2000], where inconsistency management is a primary concern.

1.3. Expected Benefits

The largest obstacle to adopting rationale capture and use as part of standard software development practices is the underlying fear that its usefulness will not be sufficient to justify the time and cost involved in its capture. Providing integrated tool support for specific uses of the rationale will help to both show how useful rationale can be as well as provide the ability to use it. In this work, we go beyond mere presentation of the rationale by using inferencing over the rationale to help ensure that design decisions are well supported and to help the software maintainer learn about the system.

While we focused on using rationale to support software maintenance, the representation of the rationale that was used to support inferencing and many of the inferences themselves are applicable to fields other than software development and will add to the general body of knowledge in the field of AI in design.

1.4. Design Rationale

Design rationale (DR) differs from other design documentation because it captures the reasoning behind the design decisions, not just capturing the final design result. This includes documenting what could have been done but was not (and why) and what had been tried but failed (and why). This information is valuable because it provides insight into the intent behind the original decisions. Documenting what was tried earlier that did not work can help prevent the maintainer from repeating mistakes made before.

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 understood 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.

We hypothesize that 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. Karsenty [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. Bratthall, et. al. [2000] performed an experiment looking specifically at 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.

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.

1.5. 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, which takes less effort to record and is easily understood by a human reader, and a formal structured representation, which 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.

1.6. The Challenge of DR in Software Engineering

The primary focus of this work is to determine when and how rationale can best be put to use. Decisions are made all throughout 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 spent the time and effort to record 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 immediately obvious what software design rationale *is*. When designing a physical object it is easy to envision choosing between alternatives such as different types of materials, colors, or shapes. For software, the obvious 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 to develop, and how easy it is to work with in the future. The type of decisions made will vary depending on the stage of the development process with the alternatives becoming more concrete as the system moves from requirements definition into implementation.

1.7. Expected Results

Our goal in this research was to investigate ways that rationale could be used during software maintenance and to build a system to support these uses that allows us to investigate using DR for software maintenance. This was done by producing the following:

1. **Uses for DR during maintenance and what has to be done with the DR to support these uses:** this was necessary in order to create a representation and system that supports maintenance.
2. **A method for using rationale to detect inconsistencies within the reasoning behind software decisions:** 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 ensure that decisions made for changes to the design and implementation are consistent with those made earlier.

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 must 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 a level of detail appropriate to the stage of development and also support the ability to compare arguments in order to evaluate the design.
5. **A way of attaching the rationale to the software implementation 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 as decisions are made and viewed by the user later when the artifacts are updated. We chose to associate rationale with the code because that is the artifact most likely to be accessed by the software maintainer.
6. **A prototype system that uses these methods to support the maintainer:** the prototype system allowed us to test out the representation and ensure that it supports the intended uses.
7. **Evaluation results for the prototype system:** the prototype system was used in an experiment to evaluate the usefulness of the system and the rationale during a series of maintenance tasks. The performance of users performing maintenance with the assistance of the system was compared with that of users who performed the same tasks without assistance.

1.8. Organization of the Dissertation

The remainder of this dissertation presents our approach to using DR to support software maintenance in more detail. Chapter 2 presents the problem we are trying to solve: using rationale to assist in software maintenance. Chapter 3 describes relevant research in design, AI in design, design rationale, and software design. Chapter 4 presents prior work in investigating DR users while Chapter 5 describes a software maintenance study that assisted in developing requirements for our rationale representation. Chapter 6 describes our approach, Chapter 7 describes the system architecture and initial design of the SEURAT system, and Chapter 8 describes the SEURAT implementation. Chapter 9 presents a demonstration of how SEURAT is used to perform several software maintenance tasks. Chapter 10 describes the evaluation of SEURAT and presents the experiment results. Chapter 11 concludes the dissertation with the conclusions of the research and some goals for future work.

As described earlier, while everyone agrees that design rationale is useful, it is still not used enough in practice. Possible 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 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?

In this chapter we discuss design rationale and its relationship to the software development process (Section 2.1), potential uses for design rationale in software development and maintenance (Section 2.2), how design rationale use could be encouraged (Section 2.3), and a summary of this chapter (Section 2.4).

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):

- *Requirements* – this phase is where the needs of the customer are examined and transformed into requirements for the resulting system.
- *Analysis* – this phase is where the requirements are analyzed and mapped on to specific functions that the system must perform.
- *Design* – this phase is where the system structure is defined based on the results of the analysis and requirements phases.

-
- *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.

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 interface was prototyped or story-boarded 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.
- *Testing* – rationale could provide the reasons behind the choice of test cases and test inputs.
- *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 2-1 shows the requirements, analysis, design, and implementation phases and the rationale that could be generated during each of them. Capturing *all* this information, however, would present a significant amount of overhead to the software developer.

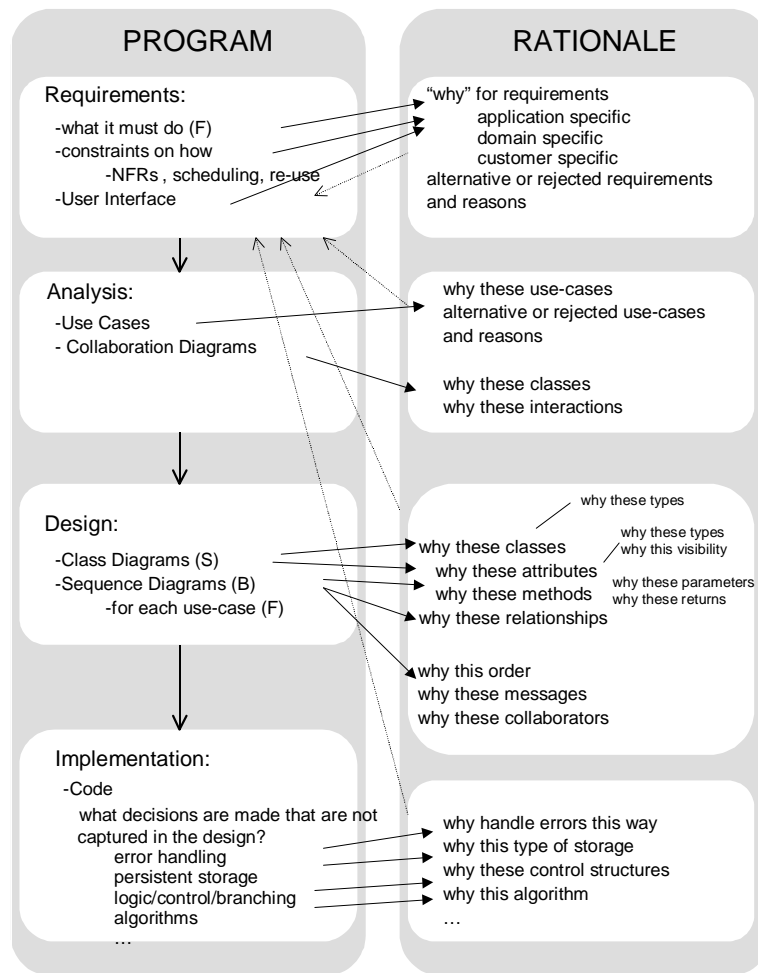


FIGURE 2-1. Software Development Phases and Rationale

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 subsequent iteration or new system design.

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. Conversely, 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 backward propagation.

2.2. Uses of Rationale in Software Development and Maintenance

There are many ways that rationale could be used in software development and maintenance. 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, particularly during maintenance where the maintainer may not even work for the same company as the original developer.

Another way that rationale 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 helps avoid repeating bad decisions and helps avoid overlooking potentially good ones. This could potentially decrease maintenance costs by helping the maintainer take advantage of what was learned earlier in the development process.

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 by 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. This will help in maintenance by giving the maintainer a way to ensure that the revisions are complete and consistent with the rest of the system.

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. Changing technology is a frequent reason why software needs to be changed during maintenance.

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.
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:

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

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 language 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. Evaluating the design will be useful during maintenance as a way to ensure that the priorities of the maintainer are consistent with those of the original developers.

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 accessible to 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. 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.

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 Sriram’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; Mittal & Frayman, 1989] 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 (tasks performed while designing) and a variety of design problems (types of things being designed).

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.

Systems generally target a particular type of design task. Some, focus on parametric design: HIPAIR [Joskowicz & Sacks, 1999] designs mechanical assemblies, DPMED [Ramachandran, et. al., 1988] performs parametric design of transmissions, DOMINIC I [Dixon et. al., 1986] is a domain independent parametric design system. Others focus on configuration design: COSSACK [Frayman & Mittal, 1987] and R1 [McDermott, 1982] perform configuration of computer systems, while VT [Marcus, et. al., 1992] designs an elevator system.

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.

Additional information about research in AI in Design can be found in [Tong & Sriram, 1999], [Brown, 1992], [Birmingham & Brown, 1997], [Stahovich, 2001], in the AIEDAM journal (Cambridge University Press), and in the collected proceedings of the AI in Design conferences.

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 understood 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 (arguments for and against alternative solutions to design decisions).

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 stated above, 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*, a claim made about 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 called 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 work that used this capture method 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 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. Another system 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.

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 rationale 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 IBIS 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. The Accord system [Ullman, 2004] uses belief networks to combine rationale in order to perform a risk analysis of concepts, or systems, being developed. This assessment shows if the best possible decisions are being made.

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 & Glagowski, 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 inductively 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. The Design-Requirements-Embedding (DRE) approach [Vanwelkenhuysen, 1995] captures rationale for a generic conceptual design. This rationale then provides assistance in modifying this pre-defined design to meet more specific needs.

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, surveyed by Dutoit and Paech [2001]. Potts and Bruns [1988] created a model of generic elements in software design rationale that was then extended by Lee [1991] in 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. REMAP [Ramesh & Dhar, 1994] extended the IBIS argumentation format to capture the rationale behind functional specifications. Zaychik and Regli [2002] developed Code-Link, a rationale support tool

that captured rationale by associating code with e-mail messages. This resulted in a context-aware e-mail system.

CoMo-Kit [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 described 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.

Sutcliffe [1995] proposed the use of rationale to aid in knowledge elicitation. He performed a study that used a prototype system and directed questioning to get requirements. One of the experimental groups asked questions intended to find design alternatives. The rationale collection promoted user participation in design decisions. The results of this study contributed to the development of the SCenario Requirements Analysis Method (SCRAM) which involved four techniques: use of a prototype that the users can work with, scenarios put that artifact in the context of its use, design rationale to expose the designers reasoning and encourage user participation, and finally a white-board

summary that identified dependencies and priorities. The rationale format used for the presentation of the results was the QOC [MacLean, et al., 1995] argument format.

Lougher and Rodden [1993] investigated maintenance rationale and built a system that attaches rationale to source code. Their approach differs from ours, however, in that they feel that maintenance rationale is very different from that captured during development and is not in the form of argumentation. We will discuss this later in this dissertation. Canfora et al. [2000] 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 (CM2) which is also based on the QOC [MacLean, et al., 1995] argumentation format.

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. Four of them are discussed here:

- software development processes;
- software requirements analysis;
- software architecture;
- software maintenance.

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] are intended to meet the primary software development goals of increased quality and reduced development costs [Osterweil, 1997].

Software development 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] suggests that software processes should be viewed 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 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] provides the definition:

“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.”

Research is being performed for a number of aspects of RE [Nuseibeh & Easterbrook, 2000]: eliciting requirements, modeling and analyzing requirements, communicating the requirements, reaching agreement on requirements, and evolving requirements. Much of the research performed within these aspects either utilizes 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 obstacles to rationale capture. These obstacles include the problems that requirements traceability is viewed as a low priority task, that rationale traceability is not managed as a priority, that rationale traceability 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. This area is related to RIM because NFRs can often conflict with each other and result in tradeoffs that must be made (cost vs. flexibility, speed vs. memory use, etc.). 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, which also give goals for the system.

One potential use of rationale is in evaluating tradeoffs. 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 tradeoffs 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 become increasingly more expensive to change as the process progresses [Pressman, 1997]. The potential impact 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.

3.4.4 Software Maintenance

Software maintenance encompasses a wide range of activities that take place after the software system has been delivered to the customer. Chapin [2000] took a wide approach when he identified twelve types of software maintenance: training, consultive, evaluative, reformative, updatative, groomative, preventive, performance, adaptive, reductive, corrective, and enhancive. Of these, the first five do not involve modifying the software. Software maintenance can be viewed as different from development because it provides a service, rather than creating a product [Niessink & Vliet, 2000]. Kitchenham, et. al. [1999] try to further define maintenance by building an ontology of factors that influence maintenance.

One system that assists in understanding large software systems (something necessary for software maintenance to be successful) is LaSSIE (Large Software System Information Environment) [Devanbu et. al., 1991] LaSSIE provides access to the software via a number of different viewpoints by making use of intelligent indexing and a domain model. This is intended to help the maintainer deal with complexity and invisibility, two of Brooks' [1995] essential difficulties in building large software systems.

Many studies have been performed to study software maintenance. The need for more attention to maintenance was demonstrated by a study performed by Hall, et al. [2001]. This showed that the formal process improvement models did not sufficiently address maintenance. Singer [1998] performed interviews at ten industrial sites. One especially interesting observation is that "source code is king", meaning that the source code is the main source of information about the system and that other documentation might be used but is not always trusted. A study of eighteen organizations in Sweden [Kajko-Mattsson,

2001] against a set of maintenance requirements (goals) showed that there was a great deal of room for improvement.

3.5 Summary

The work described in this dissertation, the SEURAT system, uses design rationale (DR) to support software maintenance. SEURAT contributes to work performed in AI in Design by defining the knowledge representation for the rationale, a semi-formal argumentation structure, and by using inference to detect errors in the rationale structure and content. The representation is similar, although an extension of, the DRL representation used in SYBIL [Lee, 1990]. Inferencing over content requires that there be a common vocabulary that allows comparison of different arguments. This need is what drove the keywords used by KBDS [King & Bañares-Alcantara, 1997] and the common vocabulary used in InfoRat [Burge & Brown, 2000]. SEURAT supports semantic inference via an argument ontology that describes a hierarchy of reasons for making software decisions that contains several levels of abstraction.

SEURAT is both related to and applicable to a number of areas in software design research. SEURAT supports requirements engineering by providing requirements traceability through using requirements as arguments for and against alternatives. SEURAT also allows rationale for requirements to be captured and used. SEURAT also can allow capture of rationale generated for the software architecture as well as decisions made elsewhere in the development process.

The main focus of SEURAT, however, is on software maintenance. The inferencing performed by SEURAT provides insight into the original intent behind the design and implementation of the software. This is information that is often unavailable in standard design documentation. SEURAT was evaluated for three types of software maintenance: adaptive, corrective, and enhancive. The results of that evaluation are presented later in this document.

SEURAT contributes to the fields of AI in Design, Software Design, and Software Maintenance through its representation, inference, argument ontology, and integration with a software development environment. Figure 3-1 shows the relationship between SEURAT and the research areas described in this chapter.

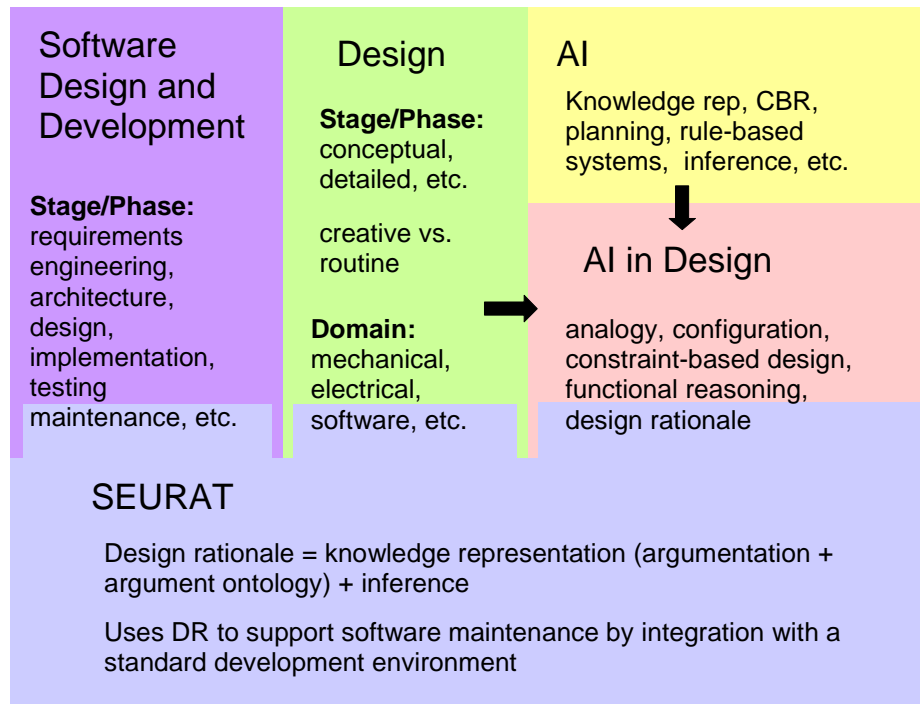


FIGURE 3-1. SEURAT Relationships

**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). This was done in preparation for building the larger system described later in the dissertation.

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: i.e., 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 4-1 shows how design rationale links the requirements and the design.

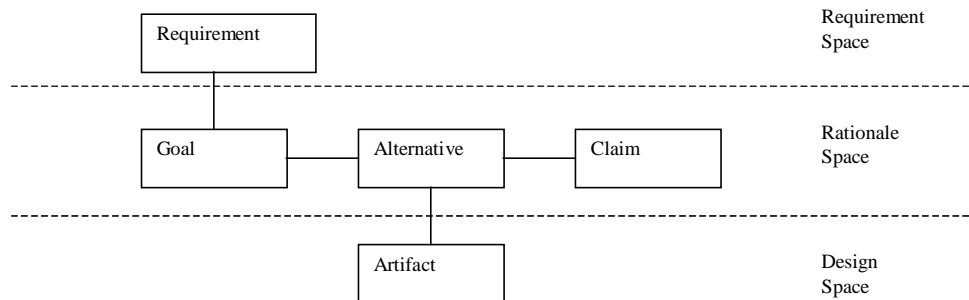


FIGURE 4-1. Design Rationale in the Design Process

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 4-2 shows the intersection.

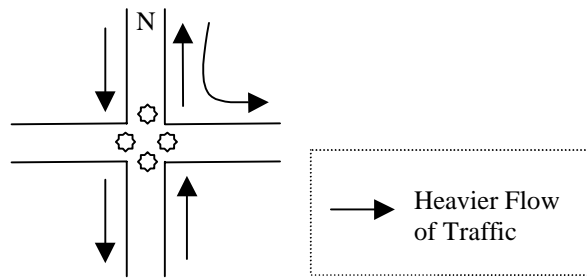


FIGURE 4-2. Intersection Diagram

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. 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 4-3 shows the elements represented in InfoRat and the relationships between them.

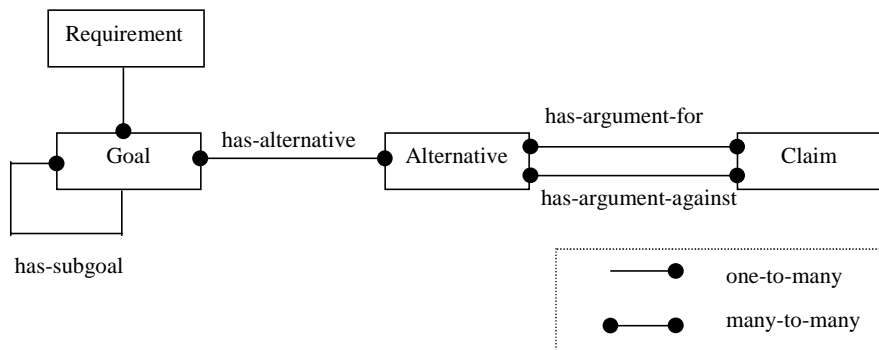


FIGURE 4-3. Design Rationale Elements

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 4-4 shows the goals as well as a partial set of alternatives and claims for the requirement to use four traffic lights.

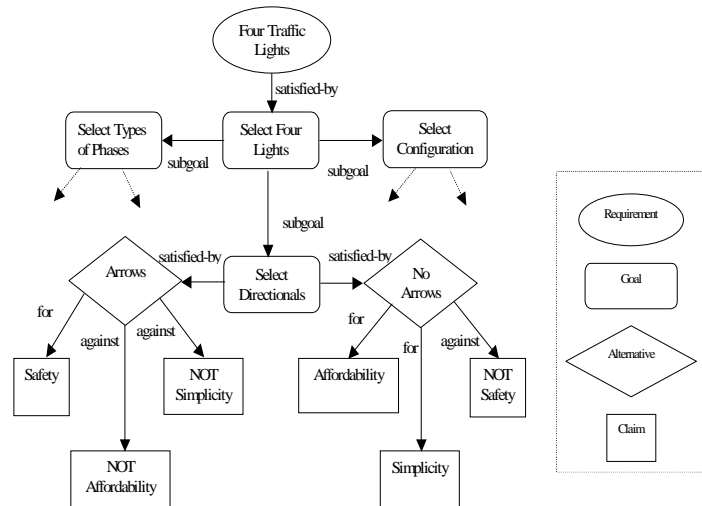


FIGURE 4-4. Subset of Alternatives for Requirement “Four Traffic Lights”

When a claim is used as an argument for or against an alternative, it is given two 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.

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 4-4) 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 0.5 and “NOT Simplicity”, with an amount of four and an importance of 0.3, its overall rating would be 3.3:

For	Against	Result
$(7 * 1) - (5 * 0.5 + 4 * 0.3) = 3.3$		

This algorithm was used because it is simple, yet is rich enough to support the types of inferencing desired for the system.

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 4-5 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 4-6 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.

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

FIGURE 4-5. Goals and Sub-goals for the Unsatisfied Requirement

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 4-7), selected alternatives where the alternative selected is not the best choice, (as shown in Figure 4-8), selected alternatives where the same argument is used both for and against the alternative, (as shown in Figure 4-9).

```

*****
*           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!

```

FIGURE 4-6. Unsatisfied Requirement Check

```

Arguments AGAINST outweigh FOR:

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

```

FIGURE 4-7. Arguments Against Outweigh For

```

Best Alternative not chosen for Select Light Configuration

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

```

FIGURE 4-8. Best Alternative Not Chosen

```
Enter Selection: 5
Same argument for and against:
  For Goal:If EW traffic, no NS traffic
    Alternative: Individual Light Control
    Claim FOR:Safety and Claim AGAINST: Safety

  For Goal:If NS traffic, no EW traffic
    Alternative: Individual Light Control
    Claim FOR:Safety and Claim AGAINST: Safety
```

FIGURE 4-9. 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 4-10, 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 4-11.

```
Rationale is not complete for Goal: Select Types of Phases

Alternative: [Italian 3-Phase Lights] is missing arguments for:
Safety
```

FIGURE 4-10. Completeness for a Goal

```
Inconsistent importance values found for Goal:
Select Types of Phases

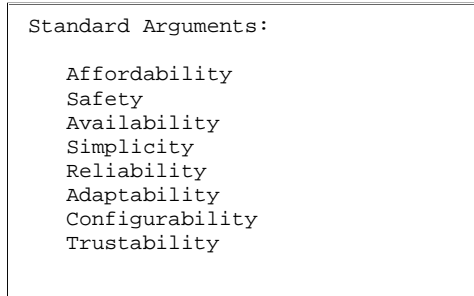
For Argument: [Availability] and Alternatives:
German 4-Phase Lights
Italian 3-Phase Lights
```

FIGURE 4-11. Consistency for a Goal

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. For the InfoRAT system, we refer to these as the Standard Claim Vocabulary and the User-Defined Claim Vocabulary respectively.

The Standard Claim Vocabulary was pre-defined to match the design task. For software design, a vocabulary was been built based on the “ilities” [Filman, 1998]. Figure 4-12 shows the Standard Claim Vocabulary used by InfoRat.

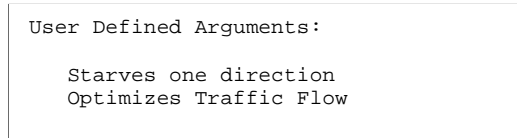


```
Standard Arguments:

    Affordability
    Safety
    Availability
    Simplicity
    Reliability
    Adaptability
    Configurability
    Trustability
```

FIGURE 4-12. Standard Claim Vocabulary

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 4-13 shows the User-Defined Claim Vocabulary for the traffic light design problem.



```
User Defined Arguments:

    Starves one direction
    Optimizes Traffic Flow
```

FIGURE 4-13. User Defined Claim Vocabulary

4.2.4 Tradeoff Evaluation

Another type of semantic inferencing makes use of background knowledge to evaluate tradeoffs. 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 4-14 shows an example of a causality error. InfoRat also checks to ensure that if two claims are causally related they never appear individually. Figure 4-15 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 4-16 shows an importance violation along with a 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 4-14. Causality Violation

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

FIGURE 4-15. Missing Claim

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 4-16. Tradeoff Importance Violation

4.3 Implementation and Examples

InfoRat was 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. Figure 4-17 through 4-19 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)

FIGURE 4-17. Requirement Listing

The remaining options give the user a more detailed view of each element. Figure 4-5 showed the information displayed about a requirement and its goals. Figure 4-20 shows the contents of an alternative, Blinking Red/Yellow.

Goals:

Select Types of Phases (Satisfied)
Select Type of Directionals (Satisfied)
Select Light Configuration (Satisfied)
If EW traffic, no NS traffic (Satisfied)
If NS traffic, no EW 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 4-18. Goal Listing

Alternatives:

German 4-Phase Lights
Italian 3-Phase Lights (Selected)
Light with Turn Signal (Selected)
Light w/o Turn Signals
All Lights the Same
Mixed Light Types (Selected)
Central Light Controller (Selected)
Individual Light Control
Blinking Red/Yellow
Sensor Controlled E/W
Configuration Changes w/Time (Selected)
Turn Arrow for S->E (Selected)
Delayed Green
All Lights go to Blinking Red (Selected)
All Lights go to Solid Red

FIGURE 4-19. Alternative Listing

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 4-21 shows an example of a version history.

```

Name: <Instance-always_blinking>
Alternative: Blinking Red/Yellow

Alternative for:
    Priority to NS Traffic (Not Selected)

Claims For:

    Claim: Simplicity
    Applicability: IS
    Amount: 3
    Importance: MODERATE

    Claim: Affordability
    Applicability: IS
    Amount: 4
    Importance: MODERATE

Claims Against:

    Claim: Safety
    Applicability: NOT
    Amount: 7
    Importance: MODERATE

```

FIGURE 4-20. Alternative Blinking Red/Yellow

```

Version History:
Version: 1

    Change: Removed claim [Safety] from
            [Configuration Changes w/Time]
    Reason: Duplicate Argument

Version: 2
    Change: Removed claim [Affordability] from
            [All Lights the Same]
    Reason: Contradiction with another argument

Version: 3
    Change: Added new Argument: [Optimizes Traffic Flow] for
            Alternative: [Mixed Light Types]
    Reason: Mixed lights can optimize flow

Version: 4
    Change: Removed claim [Safety] from
            [Individual Light Control]
    Reason: Individual lights are less safe (synch problems)

Version: 5
    Change: Changed weight of argument [Optimizes Traffic Flow]
            to 5
    Reason: Traffic flow is very important

```

FIGURE 4-21. Version History

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 4-22 shows the modification choices.

```
*****
*           Modify Design Rationale           *
*****

Choose one of the following:

1: Modify Requirements
2: Modify Goals
3: Modify Alternatives
4: Modify Arguments
5: Modify Tradeoffs

E: Exit Menu
```

FIGURE 4-22. Modify Rationale Options

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 4-23 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.

```
*****
*                               *
*       Modify Alternative       *
*                               *
*****

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
```

FIGURE 4-23. Modify Alternative Options

4.4 Summary

InfoRat was developed to demonstrate some potential uses for DR as preparation for building a larger prototype system. 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, was a first step towards providing a new and different way of looking at DR use. InfoRat demonstrated that intelligent reasoning over DR could provide more beneficial uses 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. The inferences developed when building InfoRat formed the core of the inference engine of our final prototype system. The vocabulary was a first step towards what eventually became the Argument Ontology described later in this dissertation.

One of the difficulties in studying potential uses of rationale for software design is that there are few (if any) examples of rationale 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 software design and at rationale that was generated/changed when software 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. We used a simple meeting scheduler system as the software being maintained. This system let the user enter meetings into a schedule, browse ahead and back through the schedule, and cancel meetings already scheduled.

There are a number of different classifications for types of software maintenance tasks [Chapin, 2000]. Three types were examined in this study: corrective, perfective, and enhancive.

1. *Corrective* - Corrective maintenance involves correcting failures of the system [Lientz and Swanson, 1980]. For example, 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. *Adaptive* - 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) [Chapin, 2000]. For example, 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. In retrospect, this was not an appropriate categorization because perfective maintenance typically does not change the functionality perceived by the user.

3. *Enhance* - Enhance maintenance involves replacing, adding, or extending “customer-experienced functionality” [Chapin, 2000]. For example, 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 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 5-1 shows the graphical convention used in documenting the rationale.

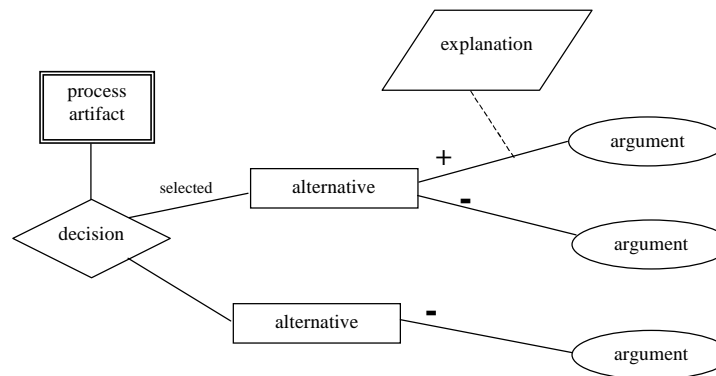


FIGURE 5-1. Rationale Components

This contained the following components:

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

During each phase of the development process, the applicable Unified Process artifacts were created along with the rationale 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 class 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 Adaptive 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 continued the artifact modifications down to the implementation.

5.2.4 Enhance 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 about rationale 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. This indicated that there needs to be a vocabulary of arguments that has different levels of abstraction so that general arguments could be used in the early phases and more specific ones used later.

The representation used in this study, with its simple +/- links for the arguments, was insufficient to express enough information to accurately document decisions. Arguments 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 was 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 Adaptive 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 Enhance 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 crucial component in determining the requirements for the rationale representation used in the system produced as part of this dissertation. In particular, it showed that the arguments needed to have detail similar to that used in InfoRat (i.e., amount and importance for each argument, not just a positive or negative direction) and that a hierarchy of common arguments would be useful so that arguments could be

expressed at a level of abstraction appropriate to the phase of the development process at which the decisions were made.

The study also showed that rationale captured during maintenance could be expressed in an argumentation format. This is different from what was posited by Lougher and Rodden [1993], who felt that maintenance rationale was not in the form of argumentation. The capture of the arguments for and against the alternatives implemented in the code (whether selected during initial development or during maintenance) is what differentiates rationale from standard comments.

In this chapter, we describe the approach we took to using rationale to support software maintenance. We feel that rationale can make maintenance both more efficient and effective and that there are uses that go beyond simply presenting the rationale to the maintainer.

In the proposal that preceded this work, we posed a number of interesting questions:

1. *How can rationale be used to assist in software maintenance?*
2. *How can decisions be captured with enough specificity to be useful yet still general enough to allow for inferencing?*
3. *Does rationale differ for different types of software modifications?*
4. *Does maintenance rationale differ from original rationale?*
5. *Are there portions of the design or phases of the design process where rationale capture would be more useful than others?*
6. *What is the relationship between rationale collected at different phases?*
7. *How can rationale changes be propagated?*

For this work, we decided to defer questions concerning rationale captured at different phases of the development process (questions 5 and 6) because we were limited to rationale that could be generated retrospectively for existing systems. For question 4, we did not see significant differences in the rationale for the different modifications. The remaining questions concentrate on rationale use during software maintenance and the representation and inferences required to support them. The uses are outlined in this chapter and the details of the representation and inferences are described later in the dissertation.

In this chapter, we review some potential uses of rationale during software maintenance (6.1), and our plan for tool support for these uses (6.2).

6.1. Uses of Rationale for Software Maintenance

As described earlier, there are many different ways that rationale can be of assistance during software development and maintenance. The rationale is often the only way to find out *why* the system was designed and implemented the way it was. Even simply *presenting* the rationale to the maintainer would be of assistance by giving them insight into the reasons behind the decisions. In this work, we expand on that by inferencing over the rationale to look for potential problems in both the rationale itself (such as incompleteness or contradictions) and the system (such as poor or inconsistent choices). Some of the uses that we wish to support are:

1. *Presentation of the rationale.* The maintainer needs to be able to access the rationale to understand the reasoning behind the decisions. This means that the rationale needs to be presented in a format that is human-readable, not just machine-interpretable.
2. *Checking for incomplete rationale.* We need to be able to detect when rationale is missing. Missing rationale could mean decisions where there were no alternatives selected, or even specified. It could also mean alternatives selected where there were no arguments. If parts of the rationale are missing, it could indicate that the decisions were not well thought out or just that the developers did not bother recording all of their reasoning. Incomplete rationale could be misleading during maintenance if the reasons behind the development of the original system are not all expressed.
3. *Checking for inconsistent rationale.* We need to insure that there are no contradictions within the rationale. This would include things like using the same argument for and against an alternative.
4. *Evaluating the support for the alternatives.* One especially important check is to evaluate the arguments for and against each alternative to see if the alternative that was chosen is the one with the most support. If a less-supported alternative is chosen that could be a poor design decision or it could mean that rationale is missing or incorrect.
5. *Expression and use of design tradeoffs.* There needs to be a way to explicitly represent and check for known tradeoffs that are made for design decisions.
6. *Propagate argument evaluations to support consistency.* There are many different reasons why some alternatives are better than others. If a reason is important for one part

of the system it is likely to be important for the rest. We need to support a way that the design priorities can be propagated through the rationale to assist in consistent decisions.

7. *Support “what-if” inferencing for changing requirements and assumptions.* With a completed system, it is often difficult to determine what the impact of changing a requirement or assumption might be. We need a way to be able to use the rationale to determine what decisions were affected by the requirements and assumptions and what parts of the system would need to be changed if a requirement or assumption is no longer valid.

These uses support the different types of software maintenance examined in this thesis: adaptive maintenance (make improvements that do not change the functionality), corrective maintenance (fixing defects), and enhanceive maintenance (adding new features).

There are a number of different things that could signal the need for adaptive maintenance. Most often, it is to improve some desirable quality that was not addressed by the initial system. For example, there may be a need to scale the system up to larger numbers of users or a larger amount of data. Rationale can assist with this by showing what decisions involved scalability as arguments for or against alternatives. The scalability improvement may involve choosing a different alternative that was already considered but not implemented because of other concerns. The more scalable alternative will have arguments for and against it that will indicate if it is a good or bad choice, and why.

In corrective maintenance, how the rationale is used will depend on the type and source of the error. If the error was the result of a decision made when designing or implementing the decision, the rationale can be used to find where the change needs to be made. The rationale may even give a better alternative for how to implement the functionality that has the problem. The rationale could also be used to point out potential problems by showing where there were decisions made that are not well supported or that violate requirements.

Enhanceive maintenance can make use of rationale in a number of ways. First, if the enhancements are related to decisions made earlier, the rationale could help point out where the code needs to be modified in order to make the enhancements. Second, the rationale can be used to check to ensure that the rationale behind the enhancements is consistent with the rationale for the rest of the system. The rationale can also be helpful in suggesting possible enhancements to the system by pointing out where some requirements and design priorities are not met.

6.2. Tool Support for Rationale Use

One key to making rationale useful is to incorporate it into the tools and process already in use for software development. This lessens the burden on the developer to remember that the rationale is present and makes it easier for them to learn how to access and use it. A tool developed to support rationale use needs to have the following characteristics:

1. *Integration with the development environment.* The developers and maintainers should not need to open up an additional tool just to create and access the rationale. The rationale needs to be made available within the tools they are already using.
2. *Association of rationale with development artifacts.* There needs to be a way to explicitly associate the rationale, particularly the alternatives, with the code that implements them.
3. *“Automatic” rationale presentation.* The maintainer should be informed or cued that rationale is available when they are working with the code that is justified by it. They should not have to deliberately search to see if there is or is not rationale associated with the code they are working on.
4. *Rationale argumentation display.* The rationale needs to be displayed in a hierarchical format that takes advantage of the structure of the argumentation. This would provide a way to view the different rationale elements in context.
5. *Automatic checking of the rationale.* The rationale status (errors and warnings generated during inferencing) should be updated automatically as rationale is added, removed, or modified.
6. *Rationale query interface.* There needs to be a way to easily search through the rationale for different rationale elements and relationships.

These tool features make the tool more useful and usable for the software maintainer and greatly increase the usefulness of rationale as an aid to software maintenance.

To drive and evaluate this research, we developed a system called Software Engineering Using RATionale (SEURAT). SEURAT supports the use of rationale during software maintenance by associating rationale with the code and by performing a series of inferences over the rationale to ensure consistency and completeness. SEURAT also provides the ability to enter the rationale into the system.

This chapter describes the design of the major functions performed by SEURAT. This consists of the SEURAT system architecture (Section 7.1), the representation (Section 7.2), the inferences performed (Section 7.3), the Argument Ontology (Section 7.4), and the plan for rationale entry and presentation (Section 7.5).

7.1. System Architecture

Figure 7-1 shows the overall system architecture for SEURAT. This shows the four major functional components of the system: the Argument Editor and Analyzer which interfaces with the user and the software system the rationale is for, the Rationale Repository, containing the rationale for the system, the Inference Engine, which performs inferencing over the rationale, and the Argument Ontology, which provides a common vocabulary for expressing the reasons for selecting alternative software design and implementation choices.

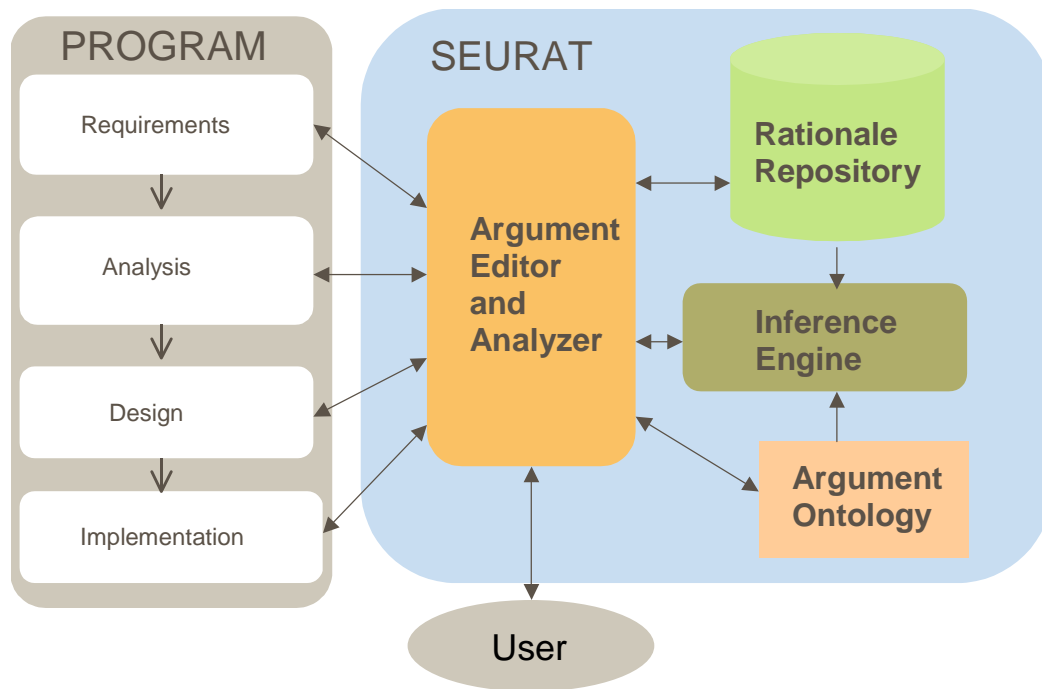


FIGURE 7-1. SEURAT System Architecture

Implementing SEURAT involved developing a representation for the rationale stored in the repository, designing and selecting inferences to perform over that rationale, defining the argument ontology, and developing ways to edit and view the rationale, to examine the results of the inferencing, and to associate the rationale with the software. The following sections describe how this was done.

7.2. Rationale Representation

This section describes RATSpeak, the Design Rationale (DR) representation used in SEURAT (Software Engineering Using RAtionale). SEURAT is being developed to allow software maintainers to use rationale when performing a variety of maintenance tasks and we needed a representation that was both flexible and expressive.

There are many different representations that have been proposed for rationale. These 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]. For SEURAT, we have taken what we feel to be the most useful features of several of these representations and incorporated them into this representation. In this document, we have chosen to express this representation using an XML schema

because it allows us to easily express the format of the rationale using simple editing tools. Using an XML schema also allows us to validate the actual rationale to ensure that it conforms to the schema, as well as allowing us to indicate which portions of the rationale are required and which are optional.

The following sections describe the elements of the SEURAT representation, how they are represented in XML, and give examples using that representation.

7.2.1. Motivation

We have chosen to represent our rationale in a semi-structured argumentation format. We feel that argumentation is the best means for expressing the advantages and disadvantages of the different software 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].

We studied these representations to learn what information could be contained in DR and to see if there was an existing representation we could use, or adapt, to meet our needs. We generated the following list of representation requirements:

1. **Support for argumentation.** We wanted to choose a representation that would express the various alternatives considered and the reasons for and against them. These reasons can be different for different alternatives for a given decision. This representation needs to include:
 - a. *Representation of alternative interactions.* We want to be able to capture interaction between alternatives to depict the case where choosing one alternative precludes or requires the choice of another. These would typically be alternatives that relate to different decisions.
 - b. *Mutually exclusive and "multiple choice" alternatives.* Some decisions may involve choosing one option, others may allow choosing multiple options.
 - c. *Arguments for and against other arguments.* In some cases, there may be disagreement with the reasons for or against selecting an alternative. For example, one designer might give an argument for an alternative saying it is safe but another designer might dispute that argument with a counter-argument giving information that states it is not safe.
 - d. *Plausibility of arguments.* There needs to be a way to indicate which arguments

are facts and which are assumptions. In the case of assumptions, the rationale needs to represent the designer's confidence in the argument.

- e. *Explicit representation of direction.* We want to be able to explicitly state the difference between an alternative being detrimental to a goal rather than simply failing to meet or contribute to it. For example, we want to make clear if an alternative is not safe (i.e., dangerous) versus having a low (or unknown) safety rating. This can be done by including a direction in the argument—explicitly saying if the alternative IS or IS NOT safe.
 - f. *Hierarchical decision representation.* Decision problems can be broken down into sub-decisions in two different ways: there are sub-decisions that result from a parent decision being broken into sub-problems that can be handled individually and there are sub-decisions that arise because a particular alternative was selected. In the latter case, the sub-decision is explicitly tied to the selection of a particular alternative. These sub-decisions describe new problems that must be solved because of the particular choice of alternative.
2. **Traceability to requirements.** We want to be able to explicitly represent which alternatives met, supported, or violated requirements. The reasons for and against an alternative do not have to be requirements but when they are it is important to represent that.
 3. **Argument ontology.** We want to be able to map the non-requirement arguments to an argument ontology in order to support semantic inferencing over the rationale.
 4. **Natural language descriptions.** We want the designer to be able to fully explain his or her choices and not be restricted to only the vocabulary provided by the ontology.
 5. **Representation of questions.** We want to be able to use the rationale to express questions that need to be answered before selecting alternatives. The rationale also needs to indicate how the question could be, or was, answered.
 6. **Design history.** We want to be able to use the rationale to generate a design history by indicating which decisions were made, or changed, at what time. This requires that the rationale be time stamped and that a version history be maintained. This version history must include the reason why the changes were made to the rationale.

7.2.2. Related Work

As described earlier, semi-formal representations are typically in the form of *argumentation*. The notation that seemed to be the most comprehensive was DRL [Lee,

1991]. For this reason, we chose to use DRL as a starting point. It was necessary to make some change 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) and because the DRL relationships between Goals, Decision Problems, and Alternatives was not always clear. We also drew on the rationale ontology work performed by Bose [1996] to give the different status values for each rationale element. Table 7-1 shows the DRL elements and how they map to our representation.

TABLE 7-1. DRL/RATSpeak Comparison

DRL	RATSpeak
Alternative	Alternative
Goal – these are the items that the alternatives support. They also map to decision problems. In DRL, a decision problem is a subtype of Goal.	RATSpeak uses Requirements and items from the Argument Ontology . The requirements will map to both the decision and to the alternative-arguments (as applicable). The argument ontology items will only map to alternative-arguments.
Decision Problem – maps to goals and a sub-type of goal.	Decision – maps to requirements or to alternatives (i.e., sometimes selecting an alternative means that we need to decide how to do it).
Claim (plausibility, evaluation, degree) – facts, assumptions, statements, or rules. Plausibility – how probable that the claim is true; Evaluation – how important the claim is; Degree – to what extent the claim is true. Evaluation is a function of degree and plausibility.	RATSpeak has split out general Claims from the arguments that indicate how they are used. The plausibility and evaluation belongs to the argument, not the claim (which will be a simple characteristic like flexibility or scalability). RATSpeak has added in the notion of importance as separate from evaluation (in RATSpeak evaluation is an overall measure taking into account the plausibility and the amount that the claim applies). RATSpeak uses importance to indicate how important it is that the alternative meet this claim/goal or that the system meet this claim/goal.

Achieves (alternative, goal)	We will use types of claims as the reasons for and against the alternatives. There will be some variant of the achieves relation for linking alternatives to requirements (via arguments for the alternative). There will also be the reverse of achieves.
Is a (good) alternative for (alternative, decision problem)	We will provide a list of alternatives considered for each decision.
Supports (claim, claim)	This would be used to support other claims.
Denies (claim, claim)	This is used to contradict a claim.
Presupposes (claim, claim) – something can be an alternative only if some other claim is true.	We use this relationship to indicate if an alternative requires that another alternative be selected.
Qualifies (claim, claim) – C1 qualifies C2 if the plausibility of C2 becomes null when that of C1 becomes low enough.	We have added an Opposes and Pre-Opposes relationship that indicates conflicts between alternatives.
Is-a-subgoal-of (goal, goal)	RATSpeak does not have an element called Goal. Instead, there are requirements, claims, and assumptions.
Is-a-subprocedure-of (procedure, procedure)	RATSpeak does not allow hierarchies of procedures. We have chosen to keep this simpler than allowed in DRL.
Answers (claim, question)	RATSpeak does not have this relationship.
Are possible answers to (group, question)	RATSpeak does not represent possible answers.
Is an answering procedure for (procedure, question)	This will be implemented in RATSpeak.
Is a result of (claim, procedure)	RATSpeak only uses procedures to answer questions. If the answer to a question affects an alternative, this will be captured by other means.
Tradeoffs (object, object, attribute)	RATSpeak allows tradeoffs to be expressed as background knowledge.

Is-a-kind-of (object, object)	There are several sub-relationships expressed in RATSpeak. There is no generic version like in DRL.
Is-a-part-of	Covered by a number of hierarchical relationships in RATSpeak (hierarchies of decisions, arguments).
Suggests (object, object)	There is no Suggests relationship in RATSpeak.
Raises (object, question)	In RATSpeak this is captured in the relationship between questions and decision problems.
Comments (claim, object)	There is no Comments relationship in RATSpeak.
Facilitates (alternative, goal)	We have a similar relationship between requirements and alternatives when an alternative facilitates meeting a requirement.
Queries (object, question)	In RATSpeak this is captured in the relationship between questions and decision problems.
Influences (question, claim) – question influences a claim if the plausibility of the claim depends on the answer to the question.	In RATSpeak, questions are attached to decision problems. The relationship between the question and specific arguments is not given explicitly although it is possible to check to make sure all questions are answered before making a decision.
Question	We will support the ability to attach questions.
Group	Groups can be created dynamically by the user creating queries on the rationale but are not expressed explicitly.
Viewpoint	Viewpoints can be created dynamically by the user creating queries on the rationale but are not expressed explicitly.

Procedure	We will support procedures.
Status	<p>RATSpeak has a status attached to the decisions, to the questions, to the alternatives, etc.</p> <p>Bose [1996] gives a good set of states, which can be attached to options, issues, and agreements. The states in RATSpeak were based on these.</p>

7.2.3. Representation Format

In order to support the SEURAT system, we created the RATSpeak representation. RATSpeak uses the following elements as part of the rationale:

1. *Requirements* – these include both functional and non-functional 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.
2. *Decision Problems* – these are the decisions that must be made as part of the development process.
3. *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).
4. *Alternatives* – these are alternative solutions to the decision problems. Each alternative will have a status that indicates if it is accepted, rejected, or pending.
5. *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).

6. *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.
7. *Assumptions* – these are similar to claims except that it is not known if they are true or will continue to hold in the future. Assumptions do not map to items in the Argument Ontology.
8. *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 a default importance that can be overridden by claims that reference it.
9. *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.

The relationships between these elements are shown in Figure 7-2. The arrows in the direction show composition relationships. For example, a decision problem has alternative solutions for the problem; each alternative has arguments for and against the decision that either relate to claims about the alternative or relationships between the alternative and requirements. Figure 7-3 shows another view of the argumentation structure of the rationale. The thin lines indicate hierarchical relationships while the thick lines indicate the type of argument.

Figure 7-4 shows the top level XML representation of the XML. This shows that the rationale is made of the Argument Ontology, requirements, decisions (also known as decision problems), and background knowledge.

```
<xsd:complexType name="Rationale">
  <xsd:sequence>
    <xsd:element name="argOntology" type="ArgOntology"
      minOccurs="1" maxOccurs="unbounded"/>
    <xsd:element name="requirement" type="Requirement"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="decisionproblem" type="DecisionProblem"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="backgroundKn" type="BackgroundKn"
      minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

FIGURE 7-4. Rationale Top Level Representation

In the following subsections, we break the XML rationale representation down into the different components and describe each of them along with an example. The example rationale is from the design of a system to play competition Solitaire.

7.2.3.1. Requirement

The requirements represent the software requirements of the system that rationale is being collected for. These requirements provide the most important reasons for making design decisions. Figure 7-5 shows the XML schema for requirements in RATSpeak.

The attributes of the requirement are an ID, used to reference the requirements in arguments, a description giving the text of the requirement, the type (Functional or Non-Functional), the artifact (used if the rationale needs to map to some other design artifacts). The requirement also has a status and arguments for and against the requirement (the requirement rationale). The rationale is given so that the reasons for and against making something a requirement are saved. This can be very important if the requirements ever need to be revisited.

Each requirement can have sub-requirements that must be met for the parent to be met. These sub-requirements are indicated by nesting them underneath the parent requirement.

```

<xsd:complexType name="Requirement">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="argument" type="Argument"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="requirement" type="Requirement"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="history" type="StatusHistory"
      minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="reqtype" type="ReqType"
    use="required"/>
  <xsd:attribute name="artifact" type="xsd:string"
    use="required"/>
  <xsd:attribute name="status" type="ReqStatus"
    use="required"/>
</xsd:complexType>

```

FIGURE 7-5. Requirement Schema

The requirement also has a status attribute defined as the type “ReqStatus” this is specifies the requirement status and is shown in Figure 7-6.

```

<xsd:simpleType name="ReqStatus">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Satisfied"/>
    <xsd:enumeration value="Violated"/>
    <xsd:enumeration value="Addressed"/>
    <xsd:enumeration value="Retracted"/>
    <xsd:enumeration value="Rejected"/>
    <xsd:enumeration value="Undecided"/>
  </xsd:restriction>
</xsd:simpleType>

```

FIGURE 7-6. Requirement Status

The requirement representation contains a history showing how the status of the item has changed over time. Each history contains a sequence of history records. Figure 7-7 shows the definitions for the status history and history records.

```
<xsd:complexType name="StatusHistory">
  <xsd:sequence>
    <xsd:element name="record" type="HistoryRecord"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="HistoryRecord">
  <xsd:attribute name="status" type="xsd:string"
    use="required"/>
  <xsd:attribute name="timestamp" type="xsd:string"
    use="required"/>
  <xsd:attribute name="reason" type="xsd:string"
    use="required"/>
</xsd:complexType>
```

FIGURE 7-7. History and History Records

Figure 7-8 shows an example of a requirement defined using this schema (the examples are displayed in color because that is how they are rendered in a web browser).

```
- <DR:requirement id="r660" name="next hand" reqtype="FR"
  artifact="" status="Undecided">
  <DR:description>A player can move to the next hand at
    any point during the game.</DR:description>
- <DR:history>
  <DR:record status="Undecided" reason="Initial Entry"
    timestamp="Sat May 03 13:53:42 EDT 2003" />
  </DR:history>
```

FIGURE 7-8. Requirement Example

7.2.3.2. Decision Problem

Decision problems describe the decisions that need to be made while designing. Figure 7-9 shows the XML schema for a decision problem.

```
<xsd:complexType name="DecisionProblem">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:choice>
      <xsd:element name="alternative" type="Alternative"
        minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="decisionproblem"
        type="DecisionProblem" minOccurs="0"
        maxOccurs="unbounded"/>
    </xsd:choice>
    <xsd:element name="question" type="Question"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="history" type="StatusHistory"
      minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="type" type="DecisionType"
    use="required"/>
  <xsd:attribute name="phase" type="Phase" use="required"/>
  <xsd:attribute name="status" type="DecisionStatus"
    use="required"/>
  <xsd:attribute name="artifact" type="xsd:string"/>
</xsd:complexType>
```

FIGURE 7-9. Decision Problem Schema

Each decision problem has a description, giving the problem that must be solved, the type of decision being made, the development phase it is made in, and a pointer to the development artifact it affects (if applicable). The decision problem consists of either a set of alternatives for that decision or a set of sub-decisions that break the problem into smaller decisions that together solve the problem. Decision problems can also have a

question associated with them and also have a status indicating if the problem has been solved or not. Figure 7-10, Figure 7-11, and Figure 7-12 give the XML schema for the decision type, the phase, and the status, respectively. The values for decision status are taken from the Issue status given in [Bose, 1996].

```
<xsd:simpleType name="DecisionType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="SingleChoice"/>
    <xsd:enumeration value="MultipleChoice"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-10. Decision Type

```
<xsd:simpleType name="Phase">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Requirements"/>
    <xsd:enumeration value="Analysis"/>
    <xsd:enumeration value="Design"/>
    <xsd:enumeration value="Implementation"/>
    <xsd:enumeration value="Test"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-11. Development Phase

```
<xsd:simpleType name="DecisionStatus">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Resolved"/>
    <xsd:enumeration value="Unresolved"/>
    <xsd:enumeration value="Non-resolvable"/>
    <xsd:enumeration value="Addressed"/>
    <xsd:enumeration value="Retracted"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-12. Decision Status

Figure 7-13 gives an example of a decision problem. This example contains arguments and alternatives, which will be described later.

```
= <DR:decisionproblem id="r675" name="How much state to
    share?" type="SingleChoice" phase="Design"
    status="Unresolved">
    <DR:description>How much state needs to be shared
        between client and server?</DR:description>
= <DR:alternative id="r677" name="Share no state"
    evaluation="0.0" status="Adopted">
    <DR:description>Share no state directly between
        clients.</DR:description>
= <DR:argument id="r678" name="State sharing is a
    bottleneck" argtype="Supports"
    plausibility="High" amount="10">
    <DR:description>"Once you start sharing state it
        just gets horrible"</DR:description>
= <DR:claim id="r680" name="Decreased Speed"
    direction="IS">
    <DR:description>Sharing state will increase
        message traffic and increase latency.
    </DR:description>
    <ref>r641</ref>
    </DR:claim>
    </DR:argument>
= <DR:history>
    <DR:record status="At_issue" reason="Initial
        Entry" datestamp="Sat May 03 14:10:20 EDT
        2003" />
    </DR:history>
    </DR:alternative>
= <DR:history>
    <DR:record status="Unresolved" reason="Initial
        Entry" datestamp="Sat May 03 14:11:35 EDT
        2003" />
    </DR:history>
</DR:decisionproblem>
```

FIGURE 7-13. Example Decision Problem

7.2.3.3. Questions

Questions can be associated with each decision problem. These are questions that need to be answered during the course of making the decision. Figure 7-14 gives the XML schema for a question.

```
<xsd:complexType name="Question">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="procedure" type="xsd:string"
      minOccurs="0" maxOccurs="1"/>
    <xsd:element name="answer" type="xsd:string" minOccurs="0"
      maxOccurs="1"/>
    <xsd:element name="history" type="StatusHistory"
      minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="status" type="QuestionStatus"
    use="required"/>
</xsd:complexType>
```

FIGURE 7-14. Question Schema

The question contains a description, which states the question. It also can contain a procedure (which could be an executable procedure or the description of how to find the answer), the answer to the question, and the status of the question. Figure 7-15 shows the XML schema for the question status. Figure 7-16 gives an example of a question.

```
<xsd:simpleType name="QuestionStatus">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Unanswered"/>
    <xsd:enumeration value="Answered"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-15. Question Status Schema

7.2.3.4. Alternatives

Alternatives are the different potential solutions to the decision problem. Figure 7-17 gives the schema for an alternative.

```
- <DR:question id="r676" name="What do clients need to
know?" status="Answered">
  <DR:description>What do the clients need to know about
each other during the game?</DR:description>
  <procedure>Ask the game developer</procedure>
  <answer>Very little - scores would be nice but not
needed</answer>
- <DR:history>
  <DR:record status="Unanswered" reason="Initial Entry"
timestamp="Sat May 03 14:07:55 EDT 2003" />
  <DR:record status="Answered" reason="Game developer
answered question." timestamp="Sat May 03 14:11:06 EDT
2003" />
</DR:history>
</DR:question>
```

FIGURE 7-16. Question Example

Each alternative contains one or more arguments for and/or against it. Alternatives also have a status to indicate if they have been selected as the answer to the decision problem (i.e. will be implemented in the software system) or not. Alternatives can also contain additional decision problems that need to be solved if the alternative is chosen.

Figure 7-18 gives the schema for the status of an alternative. The status in RATSpeak is based on the Option status described by Bose [1996].

```
<xsd:complexType name="Alternative">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="argument" type="Argument"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="history" type="StatusHistory"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="question" type="Question"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="decisionproblem"
      type="DecisionProblem" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="evaluation" type="xsd:float"
    use="required"/>
  <xsd:attribute name="status"
    type="AltStatus" use="required"/>
</xsd:complexType>
```

FIGURE 7-17. Alternative Schema

```
<xsd:simpleType name="AltStatus">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Admissible"/>
    <xsd:enumeration value="Valid"/>
    <xsd:enumeration value="Adopted"/>
    <xsd:enumeration value="At_issue"/>
    <xsd:enumeration value="Rejected"/>
    <xsd:enumeration value="Retracted"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-18. Alternative Status Schema

Figure 7-19 gives an example of an alternative. This contains the arguments for the alternative which will be described in more detail later. Also, some of the arguments have been left out for the purpose of this example.

```
- <DR:alternative id="r677" name="Share no state" evalua-
tion="0.0" status="Adopted">
  <DR:description>Share no state directly between
    clients.</DR:description>
-   <DR:argument id="r678" name="State sharing is a
    bottleneck" argtype="Supports" plausibility="High"
    amount="10">
    <DR:description>"Once you start sharing state it
      just gets horrible"</DR:description>
-   <DR:claim id="r680" name="Decreased Sped"
    direction="IS">
    <DR:description>Sharing state will increase
      message traffic and increase latency.
    </DR:description>
    <ref>r641</ref>
    </DR:claim>
  </DR:argument>
-   <DR:history>
    <DR:record status="At_issue" reason="Initial Entry"
      datestamp="Sat May 03 14:10:20 EDT 2003" />
    <DR:record status="Adopted" reason="No good reason
      why state should be shared, many reasons not to"
      datestamp="Sat May 03 14:11:33 EDT 2003" />
    </DR:history>
  </DR:alternative>
```

FIGURE 7-19. Alternative Example

7.2.3.5. Arguments

Arguments are used to contain the reasons to select, or not select, each alternative. Figure 7-20 shows the argument schema.

```

<xsd:complexType name="Argument">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:choice>
      <xsd:element name="reqRef" type="xsd:IDREF"/>
      <xsd:choice>
        <xsd:element name="ref" type="xsd:IDREF"/>
        <xsd:element name="claim" type="Claim"/>
        <xsd:element name="assumption"
          type="Assumption"/>
      </xsd:choice>
      <xsd:element name="altRef" type="xsd:IDREF"/>
    </xsd:choice>
    <xsd:element name="argument" type="Argument"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="question" type="Question"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="argtype" type="ArgType"
    use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="importance" type="Importance"
    use="optional"/>
  <xsd:attribute name="plausibility" type="Plausibility"
    use="required"/>
  <xsd:attribute name="amount" type="Amount" use="required"/>
</xsd:complexType>

```

FIGURE 7-20. Argument Schema

The argument contains a number of attributes that are used in evaluating the strength of the argument for and against the alternative. The argument type indicates which type of argument is being given. The type is what is used to determine if the argument is for or against the associated alternative. Figure 7-21 gives the schema for the argument type.

```
<xsd:simpleType name="ArgType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Supports"/>
    <xsd:enumeration value="Denies"/>
    <xsd:enumeration value="Pre-supposes"/>
    <xsd:enumeration value="Pre-supposed-by"/>
    <xsd:enumeration value="Opposes"/>
    <xsd:enumeration value="Opposed-by"/>
    <xsd:enumeration value="Addresses"/>
    <xsd:enumeration value="Satisfies"/>
    <xsd:enumeration value="Violates"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-21. Argument Type Schema

The argument type depends on whether the argument is for or against the alternative and if the argument is given by a requirement (Addresses, Satisfies, or Violates), a claim (Supports or Denies), an assumption (Supports or Denies) or an alternative (Pre-supposes or Pre-supposed-by).

Other attributes of the argument are used in determining the evaluation. These are the importance of the argument (Figure 7-22), the amount of the argument (e.g., “how safe”) (Figure 7-23), and the plausibility of the argument specifying how certain the designer is that it is true (Figure 7-24).

```
<xsd:simpleType name="Importance">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Not"/>
    <xsd:enumeration value="Low"/>
    <xsd:enumeration value="Moderate"/>
    <xsd:enumeration value="High"/>
    <xsd:enumeration value="Essential"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-22. Importance Schema

```
<xsd:simpleType name="Amount">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxExclusive value="11"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-23. Amount Schema

```
<xsd:simpleType name="Plausibility">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Low"/>
    <xsd:enumeration value="Medium"/>
    <xsd:enumeration value="High"/>
    <xsd:enumeration value="Certain"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-24. Plausibility Schema

The main body of the argument consists of one of three things: a reference to a requirement, a reference to an alternative, or either a claim or a reference to a claim. References in XML are like pointers – they are used to point towards an entity that is defined elsewhere in the document.

The reference to a requirement is used if the reason behind the argument is that it meets, supports, or violates a requirement. The reference to an argument is used if there is another alternative that either must be selected for this one to be valid or that requires that this alternative be selected in order to be valid. Other, non requirement-specific, reasons are expressed as claims. The argument can either be a reference to a claim used elsewhere or the claim itself.

Figure 7-25 shows an example of an argument.

```

- <DR:argument id="r664" name="burden on server"
  argtype="Denies" plausibility="High" amount="8">
  <DR:description>This places a heavy burden on the
    server if it has to support many players.
  </DR:description>
- <DR:claim id="r665" name="Bottleneck if many players"
  direction="IS">
  <DR:description>Causes problems if there are many
    players</DR:description>
  <ref>r230</ref>
  </DR:claim>
</DR:argument>

```

FIGURE 7-25. Argument Example

7.2.3.6. Claims

Claims are the reasons why an alternative is good or bad. Figure 7-26 shows the schema for a claim.

```

<xsd:complexType name="Claim">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="ref" type="xsd:IDREF"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="importance" type="Importance"
    use="optional"/>
  <xsd:attribute name="direction" type="Direction"
    use="required"/>
</xsd:complexType>

```

FIGURE 7-26. Claim Schema

The claim contains an ID, which is used if the claim is referenced elsewhere, a description saying what the claim does, the importance of the claim, the direction of the claim, such as

“is safe” or “is not” safe (Figure 7-27), and a reference that is a pointer to an entry in the argument ontology. This indicates a general type for the claim and is there so that semantic inferencing can be performed over the rationale. Figure 7-28 shows an example of a claim.

```
<xsd:simpleType name="Direction">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="IS"/>
    <xsd:enumeration value="NOT"/>
  </xsd:restriction>
</xsd:simpleType>
```

FIGURE 7-27. Direction Schema

```
- <DR:claim id="r680" name="Decreased Speed"
  direction="IS">
  <DR:description>Sharing state will increase message
    traffic and increase latency.</DR:description>
  <ref>r641</ref>
</DR:claim>
```

FIGURE 7-28. Claim Example

7.2.3.7. Assumptions

Assumptions are reasons for making decisions, or reasons behind arguments, that are not necessarily, or not always, true. These are kept separate from claims because they do not belong in the argument ontology. This also makes them easier to detect. Figure 7-29 shows the assumption representation. Figure 7-30 shows an example of an assumption.

7.2.3.8. Argument Ontology

The argument ontology is used to store a hierarchy of arguments used in the rationale. These are based on the “ilities” (e.g., scalability, flexibility, maintainability, usability, etc.) [Filman, 1998] but contain other general arguments as well. These allow semantic inferencing by providing a common vocabulary for the claims that can be used to look for similarities and contradictions in reasoning. Figure 7-31 gives the schema for the argument

ontology.

```
<xsd:complexType name="Assumption">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
```

FIGURE 7-29. Assumption Schema

```
- <DR:assumption id="r1" name="only start and end needed.">
  <DR:description>According to game developer - only
    start and end information is needed.
  </DR:description>
</DR:assumption>
```

FIGURE 7-30. Assumption Example

```
<xsd:complexType name="ArgOntology">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="ref" type="xsd:IDREF"/>
      <xsd:element name="ontEntry" type="OntologyEntry"
        minOccurs="0" maxOccurs="unbounded"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

FIGURE 7-31. Argument Ontology Schema

The ontology consists of a hierarchy of ontology entries. Figure 7-32 gives the schema for an ontology entry.

```
<xsd:complexType name="Entry">
  <xsd:choice>
    <xsd:element name="ref" type="xsd:IDREF" minOccurs="0"
      maxOccurs="1"/>
    <xsd:element name="ontEntry" type="OntologyEntry"
      minOccurs="0" maxOccurs="1"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="OntologyEntry">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="subEntry" type="Entry" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="importance" type="Importance"
    use="required"/>
</xsd:complexType>
```

FIGURE 7-32. Ontology Entry Schema

Each ontology entry contains a description, an ID, and any ontology entries beneath it in the hierarchy. The ID is necessary so that the rest of the rationale can reference it. Figure 7-33 gives an example of an argument ontology.

```

- <DR:argOntology>
-   <DR:ontEntry id="r0" name="Argument-Ontology"
-     importance="Moderate">
-     <DR:description />
-     <DR:subEntry>
-       <DR:ontEntry id="r2" name="Affordability
-         Criteria" importance="Moderate">
-         <DR:description>These arguments refer to the
-           cost of the software</DR:description>
-         <DR:subEntry>
-           <DR:ontEntry id="r4" name="Development
-             Cost" importance="Moderate">
-             <DR:description />
-           </DR:ontEntry>
-         </DR:subEntry>
-       </DR:ontEntry>
-     </DR:subEntry>
-   </DR:ontEntry>
- </DR:argOntology>

```

FIGURE 7-33. Argument Ontology Example

7.2.3.9. Background Knowledge

The background knowledge gives any tradeoffs and co-occurrence relationships that are likely to apply to the design as a whole. Figure 7-34 shows the XML schema representation for the background knowledge.

```

<xsd:complexType name="BackgroundKn">
  <xsd:sequence>
    <xsd:element name="tradeoff" type="Tradeoff"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="co-occurrence" type="CoOccurrence"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

```

FIGURE 7-34. Background Knowledge Schema

The two types of relationships shown are tradeoffs and co-occurrence relationships.

Figure 7-35 and Figure 7-36 show the schemas describing the tradeoffs and co-occurrences.

```
<xsd:complexType name="Tradeoff">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="ref" type="xsd:IDREF" minOccurs="2"
      maxOccurs="2"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="symmetric" type="xsd:boolean"
    use="required"/>
</xsd:complexType>
```

FIGURE 7-35. Tradeoff Schema

```
<xsd:complexType name="CoOccurrence">
  <xsd:sequence>
    <xsd:element name="description" type="xsd:string"
      minOccurs="1" maxOccurs="1"/>
    <xsd:element name="ref" type="xsd:IDREF" minOccurs="2"
      maxOccurs="2"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="name" type="xsd:string"
    use="required"/>
  <xsd:attribute name="symmetric" type="xsd:boolean"
    use="required"/>
</xsd:complexType>
```

FIGURE 7-36. Co-Occurrence Relationship Schema

Figure 7-37 shows an example of background knowledge.

```
<DR:backgroundKn>
<DR:tradeoff id="r687" name="flexibility vs. cost" symmetric="false">
  <DR:description>A more flexible solution costs more to deve-
lope</DR:description>
  <ref>r194</ref>
  <ref>r4</ref>
</DR:tradeoff>
</DR:backgroundKn>
```

FIGURE 7-37. Background Knowledge Example

7.3. Inferences Supported

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. It can provide even more useful information about the design if there is a way to perform inferences over it. In the following sections we describe a number of different inferences that could be performed over rationale that is structured using the RATSpeak representation.

A specific inference may or may not directly correspond to a specific use for the rationale. For example, the rationale could be used to see if any decisions were made that violated requirements. That would require inferencing that looked at the selected alternatives to see if there were any arguments against them that referred to violated requirements. In that case, there is a direct correspondence between the use, checking for requirements violations, and the inference. On the other hand, there could be a general rationale validation that is performed using multiple inferences.

De-coupling the description of the inference from how it is used encourages flexibility in the different types of uses that could be made of the rationale. For this reason, the focus of this chapter will be on the inferences themselves, not their use.

There are a number of different ways that the inferences can be categorized. The two main groups are syntactic and semantic. 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.

Another categorization is between inferences that require background knowledge and those that do not. Ideally, all inference would be stand-alone, i.e., utilizing only information that is inside the rationale. This could be done by mining rationale for

interesting patterns to be used in inference, such as common tradeoffs. This is not an easy task and is beyond the scope of this work. Instead, there the background knowledge will supply the patterns that need to be checked.

A third category is inferences that concern changes in the rationale over time. These require that the history of changes to the rationale be maintained somehow.

7.3.1. Syntactic

As described earlier, syntactic inferencing looks mostly at the structure of the rationale. The following sections describe the syntactic inferences used in SEURAT.

7.3.1.1. No Reason For Selection

Purpose: This applies when an alternative has been selected where there are no arguments for the alternative. Used to detect when a poor decision has been made. This could either indicate that the wrong alternative was selected or that the rationale is not complete. In either case, this is an error and should be reported.

Checked: This is checked automatically when an alternative is selected.

Applies to: Selected Alternatives

Level: Violation

7.3.1.2. Selection Contradicted

Purpose: This applies when an alternative has been selected where there are arguments against the alternative but no arguments for the alternative. Used to detect when a poor decision has been made. This could either indicate that the wrong alternative was selected or that the rationale is not complete. In either case, this is an error and should be reported.

Checked: This is checked automatically when an alternative is selected.

Applies to: Selected Alternatives

Level: Violation

7.3.1.3. No Selected Alternative

Purpose: Indicates that the rationale and/or the design is not complete. In this case, there is a decision where a selected alternative has not been specified.

Checked: This is checked as soon as the decision is created. This means that there will be errors reported because the rationale is incomplete, but that is considered an advantage because the error messages remind the developer that

they need to make the decision.
Applies to: Decision
Level: Violation

7.3.1.4. Too Many Selected Alternatives

Purpose: Indicates that two alternatives have been selected for a decision when the decision is only supposed to have one.
Checked: This is checked whenever any of the alternatives is edited.
Applies to: Decision
Level: Violation

7.3.1.5. Missing Sub-Decisions

Purpose: Indicates that a decision which is supposed to be expanded by sub-decisions does not have any sub-decisions listed in the rationale. This indicates that the rationale is incomplete.
Checked: This is checked whenever a decision is edited or created. The check is made for the updated decision and, if applicable, its parent decision.
Applies to: Decision
Level: Violation

7.3.1.6. Unanswered Questions

Purpose: Indicates that there is a question in the rationale that has not been answered.
Checked: This is checked as soon as the question is created and will remind the developer that they need to answer the question in order to make their decision.
Applies to: Decision or Alternative
Level: Warning

7.3.2. Semantic

Semantic inferences look inside the structure of the rationale to look for errors in the contents of a particular argument.

7.3.2.1. Best Supported Alternative not Selected

Purpose: Indicates that the alternative that was selected is not as well supported as one or more of the other alternatives for the decision.
Checked: This is checked as soon as an alternative is selected for the decision. It is

also checked whenever any of the arguments for any of the alternatives for the decision are modified. This allows detection of problems introduced if a requirement or assumption is disabled or if new arguments are added.

Applies to: Decision

Level: Warning

7.3.2.2. Requirements Violation

Purpose: Indicates that an alternative was selected that violates a requirement.

Checked: This is checked each time an alternative and/or its arguments are edited. This includes a check whenever an alternative is selected.

Applies to: Alternative and Requirement

Level: Error

7.3.2.3. Contradictory Arguments

Purpose: This looks to see if the same argument is used both for and against an alternative. This indicates a probable error in the rationale.

Checked: This is checked whenever an alternative or one of its arguments is modified.

Applies to: Alternative

Level: Violation

7.3.2.4. Duplicate Arguments

Purpose: This looks to see if the same argument is used multiple times for an alternative. This indicates a probable error in the rationale.

Checked: This is checked whenever an alternative or one of its arguments is modified.

Applies to: Alternative

Level: Violation

7.3.2.5. Pre-Supposed Not Selected

Purpose: This looks to see if there is an alternative that requires that another alternative is selected that is not (i.e., the first alternative pre-supposes the latter). This is done by checking in both directions (making sure the alternative does not pre-suppose another and that no alternatives pre-suppose it).

Checked: This is checked whenever a selected alternative or one of its arguments is

Applies to: modified to ensure that it does not depend on a non-selected alternative.
Level: Alternative
Violation

7.3.2.6. Opposed Selected

Purpose: This looks to see if there is an alternative that requires that another alternative must not be selected.

Checked: This is checked whenever a selected alternative or one of its arguments is modified to ensure that it is not opposed by another alternative is selected. This is done by checking in both directions to make sure that each alternative is not opposed by, or in opposition to, another selected alternative.

Applies to: Alternative
Level: Violation

7.3.2.7. Co-occurrence Relationship Violation

Purpose: Makes use of background knowledge to detect arguments that have co-occurrence relationships. In this case, these arguments must go together on the same side of the argument and a violation is present if they are on opposite sides instead.

Checked: This is checked whenever arguments are edited.

Applies to: Decision
Level: Warning

7.3.2.8. Co-Occurrence Relationship Missing

Purpose: Makes use of background knowledge to detect arguments that have co-occurrence relationships. In this case, these arguments must go together on the same side of the argument and should both be present.

Checked: This is checked whenever arguments are edited.

Applies to: Decision
Level: Warning

7.3.2.9. Tradeoff Violation

Purpose: Indicates that a known tradeoff (as specified in background knowledge) was violated in the rationale. This occurs when traded off items were on the same side of the argument when normally they are opposing. This could

point out an error in the rationale.
Checked: This is checked whenever arguments are edited.
Applies to: Decision
Level: Warning

7.3.2.10. Tradeoff Missing

Purpose: Indicates that a known tradeoff (as specified in background knowledge) was not taken into account in the rationale. This occurs when only one side of the tradeoff is accounted for in the rationale. This could indicate that the rationale is incomplete because the opposing arguments are not present.

Checked: This is checked whenever arguments are edited.
Applies to: Decision
Level: Warning

7.3.3. Queries

Rationale queries are used to get information about the rationale on request. The following sections describe some queries supported by SEURAT.

7.3.3.1. Find Entity

The Find Entity query allows the user to search for a rationale entity of a particular type and edit that entity. This allows the system to be more scalable to large amounts of rationale.

7.3.3.2. Find Requirement

The Find Requirement query lets the user search for requirements with a specific status: satisfied, addressed, or violated. This status only applies to requirements that relate to selected alternatives – if an requirement is violated by an alternative that is not selected it does not count as a violation.

7.3.3.3. Find Common Arguments

The Find Common Arguments query gives a list of either claims, arguments, or argument ontology entries that are sorted by the number of times the item appears in the rationale. This is useful to see if what criteria were the most involved in making design decisions.

7.3.3.4. Find Status Overrides

The Find Status Overrides query shows a list of any status messages that were overridden by the user. The user is allowed to override any of the error and warning messages

displayed by SEURAT if they feel that these errors are not reflective of the true state of the system and/or the rationale. The overrides can be easily removed, however, from the results returned by the Find Status Overrides query.

7.3.3.5. Find Importance Overrides

The user can assign importance values to arguments or claims or choose to inherit the importance of the item below it (argument ontology entries in the case of claims and claims in the case of arguments). The Find Importance Overrides gives a list of what has been overridden for each of these three types of items. The user can then choose to edit the item and remove the override. It is important to know when values are being overridden because this rationale will not be affected by any global importance changes in claims or ontology entries.

7.3.3.6. Find Related Arguments

The Find Related Arguments query can be run on any requirement to look for alternatives that satisfy, address, or violate the requirement. This is a helpful way to determine what is affected by any given requirement.

7.3.4. Historical

Historical inferences make use of a history of rationale changes that is maintained whenever the status of a rationale item changes. This provides additional insight into how the design changes over time.

7.3.4.1. Detecting Rejected Alternatives

SEURAT will alert the user if they try to select an alternative that was rejected earlier. This is intended to help prevent earlier mistakes from being inadvertently repeated.

7.3.4.2. Element History

A history of all the status changes can be shown for each requirement, decision, alternative, and question. This can show how the rationale has been changed and why over time.

7.4. 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” [Filman, 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, which provided many elements for the ontology, as well as specific criteria for NFRs relating to performance and auditing.

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. The high level criteria were as follows:

- Affordability Criteria;
- Adaptability Criteria;
- Dependability Criteria;
- End User Criteria;
- Needs Satisfaction Criteria;
- Maintainability Criteria;
- Performance Criteria.

Each of these criteria then have sub-criteria at increasingly more detailed levels. As an example. The more detailed 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, e.g., "supports". 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.

One thing to note is that the ontology 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 following sections present the Argument Ontology entries for each of the high level categories.

7.4.1. Affordability Criteria

Affordability refers to the cost to develop the software system. This is a non-functional requirement that is great concern to the software developer, manager, and customer. The affordability criteria given in the ontology are divided into the five major categories given by Bruegge and Dutoit [2000]: Development cost, Deployment cost, Upgrade cost, Maintenance cost, Administration cost. More detailed ways that cost can be reduced are listed

under these categories. Many of these involve time since time generally translates into money. Most of these come from Chung, et al. [2000]. Figure 7-38 shows the affordability criteria in SEURAT.

7.4.2. Adaptability Criteria

Adaptability refers to how easy it is to modify the software to adapt it to changing circumstances or to add new functionality. Adaptability includes two particularly crucial sub-criteria: portability and scalability. Scalability is one of the elements in the ontology that has sub-elements that also appear elsewhere in the ontology. This is because scalability is often linked to performance. Many of the scalability criteria came from Bondi [2000]. Figure 7-39 shows the adaptability criteria.

<p>Development Cost</p> <ul style="list-style-type: none"> • Uses Standard Tools and Environments <ul style="list-style-type: none"> • {is a uses a} standard development tool(s) • {is a uses a} standard language • Uses Familiar Tools and Environments <ul style="list-style-type: none"> • {is a uses a} familiar language • utilizes developer experience • {is a uses a} familiar development environment • {is a uses a} familiar hardware platform • Reduces Development Time <ul style="list-style-type: none"> • is component based • uses COTS/GOTS software • reduces customization • utilizes existing code developed in-house • uses automatically generated code • Reduces Project Success Risk <ul style="list-style-type: none"> • {is a uses a} mature language • {is a uses a} mature process • Reduces Prototyping Cost <ul style="list-style-type: none"> • reduces prototyping time • Reduces Risk Analysis Cost <ul style="list-style-type: none"> • reduces risk analysis time • Reduces Component Integration Cost <ul style="list-style-type: none"> • reduces component integration time • Reduces Domain Analysis Cost <ul style="list-style-type: none"> • reduces domain analysis time • Reduces Inspection Cost <ul style="list-style-type: none"> • reduces inspection time <p>Operating Cost</p> <ul style="list-style-type: none"> • Minimizes Communication Cost 	<p>Deployment Cost</p> <ul style="list-style-type: none"> • Minimizes Equipment Cost <ul style="list-style-type: none"> • reduces hardware cost • Minimizes External Software Cost <ul style="list-style-type: none"> • {is uses} open source • Minimizes Deployment Time <ul style="list-style-type: none"> • reduces software production time • reduces customer evaluation time <p>Maintenance Cost</p> <ul style="list-style-type: none"> • Reduces Maintenance Time <ul style="list-style-type: none"> • reduces re-compilation • Reduces Support Cost <ul style="list-style-type: none"> • increases hardware support available • increases software support available • Reduces Re-engineering Cost • Reduces Retirement Cost <p>Upgrade Cost</p> <ul style="list-style-type: none"> • Reduces COTS Risk <ul style="list-style-type: none"> • isolates code dependent on outside software • reduces vendor dependencies • reduces version dependencies • isolates version dependencies <p>Administration Cost</p> <ul style="list-style-type: none"> • Reduces Coordination Cost <ul style="list-style-type: none"> • reduces coordination time • Reduces Planning Cost <ul style="list-style-type: none"> • reduces planning time • Reduces Project Tracking Cost • Reduces Process Management Cost <ul style="list-style-type: none"> • reduces process management time
---	---

FIGURE 7-38. Affordability Criteria

<p>Extensibility</p> <ul style="list-style-type: none">• Minimizes Modification Impact<ul style="list-style-type: none">• isolates likely to change code• reduces modification impact• reduces change coordination• facilitates wrappability• uses replaceable modules• Minimizes the Amount of Code to Modify<ul style="list-style-type: none">• increases commonality• reduces coupling• increases encapsulation• increases cohesion• Simplifies Modification<ul style="list-style-type: none">• uses a design pattern• reduces duplication• provides modularity• provides information hiding <p>Modifiability</p> <ul style="list-style-type: none">• Increases Flexibility<ul style="list-style-type: none">• {provides supports} reflection• provides tunable parameters <p>Adaptability</p> <ul style="list-style-type: none">• Increases Additivity• Increases Elasticity• Increases Composibility <p>Portability</p> <ul style="list-style-type: none">• Reduces Hardware Dependencies<ul style="list-style-type: none">• isolates hardware dependent code• Reduces Software Dependencies<ul style="list-style-type: none">• {avoids reduces} OS dependencies	<p>Scalability</p> <ul style="list-style-type: none">• Increases Scalability<ul style="list-style-type: none">• {allows supports} additional users• {provides supports} policy/mechanism separation• adapts to increase in intensity of use• minimizes connections to be set up• supports functionality reuse• avoids fixed data sizes• Response Time and Throughput (see Performance)• Memory Efficiency (see Performance) <p>Reusability</p> <p>Interoperability</p> <ul style="list-style-type: none">• Provides Interface Standardization<ul style="list-style-type: none">• {is a uses a} defined interface• {is a uses a} standard interface• conforms to an API• {provides supports} consistent interfaces• {is a conforms to a} standard protocol• Supports Easier Integration<ul style="list-style-type: none">• exposes the API• reduces shared data• provides compatibility
---	---

FIGURE 7-39. Adaptability Criteria

7.4.3. Dependability Criteria

Dependability criteria include the many factors that contribute to a system being dependable. This includes security, which contains many criteria from Chung, et. al. [2000], fault tolerance [Siewiorek, 1990], and safety [Rushby, 1994]. Figure 7-40 shows the dependability criteria.

<p>Security</p> <ul style="list-style-type: none"> • Provides Access Control <ul style="list-style-type: none"> • require authorization • {provides supports} multiple authorization/access levels • {provides supports} mandatory access controls • {provides supports} discretionary access controls • Increases Data Security <ul style="list-style-type: none"> • {provides supports} data encryption • {provides supports} network isolation • Responds to Threats <ul style="list-style-type: none"> • {provides supports} countermeasures • prevents denial of service • {provides supports} threat detection • {provides supports} threat prevention • {provides supports} threat recovery <p>Robustness</p> <ul style="list-style-type: none"> • Responds to User Error <ul style="list-style-type: none"> • prevents user error • minimizes user error • detects user error • recovers from user error • requests action confirmation • requires action confirmation <p>Availability</p> <ul style="list-style-type: none"> • Reduces Error Rates 	<p>Fault Tolerance</p> <ul style="list-style-type: none"> • Handles Faults <ul style="list-style-type: none"> • {provides supports} graceful degradation • {provides supports} replication • {provides supports} failover • {provides supports} fault masking • {provides supports} retry when failure • {provides supports} restart when failure • {provides supports} reconfigure when failure • {provides supports} failure repair • provides recovery blocks • Tolerates Faults <ul style="list-style-type: none"> • {provides supports} data recoverability • {provides supports} state recoverability • {provides supports} fault detection • {provides supports} fault confinement <p>Reliability</p> <ul style="list-style-type: none"> • Prevents Data Loss <ul style="list-style-type: none"> • {is a supports a} reliable protocol • prevents data overwrites <p>Safety</p> <ul style="list-style-type: none"> • Increases Maturity <ul style="list-style-type: none"> • {is an uses an} evaluated technology • Increases Predictability <ul style="list-style-type: none"> • provides stability • provides a contract
--	--

FIGURE 7-40. Dependability Criteria

7.4.4. End User Criteria

End User Criteria give reasons for why the software system assists the user. This falls into two major categories: usability and integrity. Many of the usability criteria originated in an extensive list from Dix, et. al. [1998]. Chung, et. al, [2000] provided detailed criteria that contribute to accuracy. Figure 7-41 shows the end user criteria.

<p>Usability</p> <ul style="list-style-type: none"> • Increases Physical Ease of Use <ul style="list-style-type: none"> • {provides supports} effective use of screen real-estate • minimizes keystrokes • {provides supports} increased visual contrast • is easy to read • Increases Cognitive Ease of Use <ul style="list-style-type: none"> • provides reasonable default values • provides user guidance • {encourages supports} direct manipulation • minimizes memory load on user • provides feedback • {conforms to utilizes} user experience • increased visibility of function to users • uses predictable sequences • intuitiveness • {provides an supports an} appropriate metaphor • Increases Interface Consistency <ul style="list-style-type: none"> • {provides supports} data entry consistency • {provides supports} data display consistency • {provides supports} color and style consistency • Increases Recoverability <ul style="list-style-type: none"> • supports undo of user actions • corrects user errors • prevents user error • Increases Learnability • Increases Acceptability <ul style="list-style-type: none"> • increases aesthetic value • avoids offensiveness 	<ul style="list-style-type: none"> • Provides User Customization <ul style="list-style-type: none"> • {provides supports} customization • supports different levels of user expertise • Supports Internationalization <ul style="list-style-type: none"> • Reduces cultural dependencies • Increases Accessibility <ul style="list-style-type: none"> • Visual accessibility • Auditory accessibility • Mobility accessibility • Cognitive accessibility <p>Integrity</p> <ul style="list-style-type: none"> • Increases Completeness • Increases Consistency <ul style="list-style-type: none"> • {provides supports} internal consistency • {provides supports} external consistency • Increases Accuracy <ul style="list-style-type: none"> • {provides supports} exception handling • supports resource assignment • provides validation • provides justification enforcement • provides verification • {provides supports} a checkpoint • {provides supports} better information flow • {provides supports} authentication enforcement • {provides supports} auditing • {provides supports} consistency checking • requests confirmation • performs cross examination • provides tracking assistance • provides certification • requests authorization • provides precision
---	---

FIGURE 7-41. End User Criteria

7.4.5. Needs Satisfaction Criteria

Needs Satisfaction Criteria refer to the support given to ensure that the software meets the needs of the user. This category name came from the CMU's Quality Measures Taxonomy [SEI, 2000] but many of the sub-categories from that taxonomy ended up in different categories for the Argument Ontology. The category that remains, includes criteria from Voas and Miller [1995]. Figure 7-42 shows the needs satisfaction criteria.

Verifiability <ul style="list-style-type: none">• Increases Testability<ul style="list-style-type: none">• increases visibility of function to be evaluated• supports instrumentation• provides re-entry points• provides triggers	<ul style="list-style-type: none">• minimizes variable reuse• supports internal information capture• facilitates repeatability <ul style="list-style-type: none">• Increases Auditability
--	---

FIGURE 7-42. Needs Satisfaction Criteria

7.4.6. Maintainability Criteria

These criteria give ways that the software can be easier to maintain. The major categories are readability, supportability, and traceability. No sub-categories have been determined for supportability and traceability. Figure 7-43 shows the maintainability criteria.

Readability <ul style="list-style-type: none">• Increases Code Understandability<ul style="list-style-type: none">• Provides good documentation• {provides supports} code consistency• {provides supports} consistent method naming	<ul style="list-style-type: none">• {provides supports} code readability• {provides supports} decomposability Supportability Traceability
--	---

FIGURE 7-43. Maintainability Criteria

7.4.7. Performance Criteria

These criteria give ways that performance can be improved. Most performance criteria come from Bondi [2000] (in the context of scalability) and from Chung, et. al. [2000]. Figure 7-44 shows the performance criteria.

Response Time and Throughput <ul style="list-style-type: none"> • Increases Speed <ul style="list-style-type: none"> • {provides supports} distribution • {provides supports} parallelism • {provides supports} congestion control • {provides supports} efficient resource scheduling • {provides supports} caching • {provides supports} load shedding • {provides supports} multi-threading • {is a uses a} fast language • {is a uses a} efficient algorithm • Reduces Latency <ul style="list-style-type: none"> • {provides supports} increased processing speed • decreases latency/perceived delay 	<ul style="list-style-type: none"> • Minimizes Resource Conflicts <ul style="list-style-type: none"> • avoids deadlock • avoids starvation • minimizes contention • Optimizes Resource Use <ul style="list-style-type: none"> • {provides supports} increased component capacity • reduces component load • minimizes bandwidth • minimizes persistent storage • {provides supports} bandwidth change adaptation Memory Efficiency <ul style="list-style-type: none"> • Minimizes Memory Use <ul style="list-style-type: none"> • avoids paging • prevents memory leaks • minimizes secondary storage use
--	---

FIGURE 7-44. Performance Criteria

7.5. Rationale Entry and Presentation

The ability to view and edit the rationale is a crucial component to any rationale support system. One of the concerns about capturing rationale is the possibility that the rationale will not be consulted even when it is present. In order to encourage this, a primary goal was to provide a way to know if rationale was present while editing the code.

The following requirements were developed for the editing and presentation of the rationale:

- Explicit rationale to code associations where the user would be informed that there was rationale for the code that he or she was editing;
- Visualization of the rationale structure by showing the argumentation in a hierarchical form;
- Easy entry and modification of the rationale items.

To meet these requirements, we designed an interface where the rationale would be stored in a tree format where expanding a decision node would show the alternatives beneath it, expanding an alternative would show the arguments beneath it, etc. This tree would also

support editing of the rationale by giving the user a way to open an editor to display and modify the rationale by double-clicking on the item in the tree.

The ability of the maintainer to know if there was rationale present was a more difficult problem. This required integrating the rationale display with an editor that could be used to modify the code. This problem was solved when the Eclipse framework was selected for implementation of SEURAT. Eclipse, and how it was used to associate rationale with code, is described in the chapter on implementation.

SEURAT was developed as an Eclipse Plugin (www.eclipse.org). Eclipse is an open-source development framework that provides (among other things) a Java IDE. This allows the SEURAT capabilities (rationale presentation and use) to be available within the same development environment used by the software maintainer. This is a key feature to our approach and one that has been lacking in most rationale support tools. All of the SEURAT implementation was written using Java.

This chapter describes the software design and implementation of the SEURAT system. Section 8.1 presents the SEURAT software architecture. Section 8.2 describes the design and implementation of the Rationale Repository. Section 8.3 describes the Rationale Explorer, Section 8.4 describes how tasks are determined (via inference) and presented in SEURAT, and Section 8.5 talks about the way the code and rationale are connected. Section 8.6 describes the Rationale Editor/Viewer capability and Section 8.7 describes the Rationale Query implementation.

8.1. SEURAT Software Architecture

The previous chapter presented the system architecture for the SEURAT system which showed the main functions of the system: the Argument Editor and Analyzer, the Inference Engine, the Rationale Repository, and the Argument Ontology. The SEURAT software architecture maps these functions on to the different displays and software components that were developed within the Eclipse Framework. Figure 8-1 shows the Software Architecture.

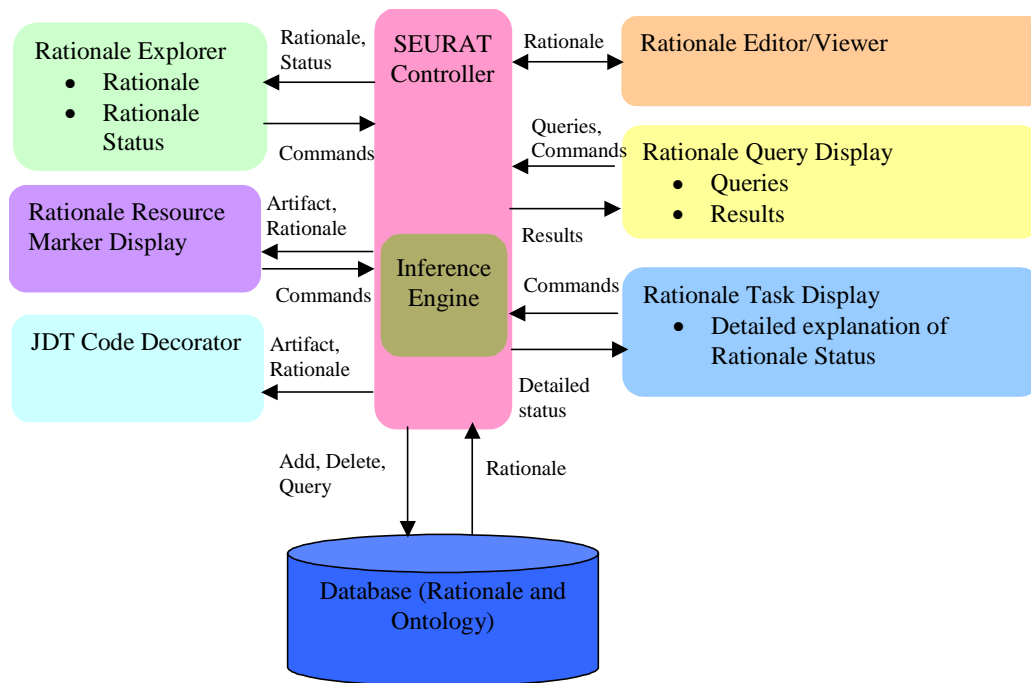


FIGURE 8-1. SEURAT Software Architecture

This diagram contains the following parts:

- *SEURAT Controller and Inference Engine* – this serves as the main point of control for the SEURAT application. It performs any inferencing needed to maintain the rationale status and respond to rationale queries.
- *Rationale Explorer* – this will give a tree-format view into the Rationale. It will serve two main purposes in the SEURAT system: it provides a way to access the rationale by selecting rationale items in the tree and editing them, and also provides a high-level picture of the rationale status. The status will be indicated by small graphical status indicators that will appear in the tree next to the affected rationale elements. The icons used to indicate errors and warnings on the rationale are the same as those used to indicate errors and warnings within the code.
- *Rationale Resource Marker Display* – there are several ways that the associations between the rationale and the code are displayed. One is by having a “decorator” icon overlay the icon for the code associated with the rationale. This is shown in the Eclipse package explorer. Another is by putting a “bookmark” on the associated code – this is

shown by having the bookmark added to a list of bookmarks maintained by Eclipse and by having a bookmark indicator shown next to the code when it is edited. When the maintainer puts their mouse over this indicator, the applicable rationale is displayed.

- *Rationale Editor/Viewer* – this is the primary means of editing and viewing elements of rationale. This can be invoked in a variety of ways, the primary one via the Rationale Explorer.
- *Rationale Query Display* – this is the means for querying the rationale for specific combinations of information. It will have an interface that allows the user to build common types of queries. The results will be a list of rationale elements that meet the query criteria. This list can then be used to invoke the Rationale Editor/Viewer to look at specific elements.
- *Rationale Task Display* – this display provides a list and description of any problems (errors or warnings) with the rationale. These are listed as tasks that the SEURAT user should consider doing. This is analogous to the way that compilation errors and warnings are displayed by Eclipse. The display will be updated automatically when the rationale changes. The user can use this list to invoke the Rationale Editor/Viewer to look at the elements causing the problem.

The flow-chart in Figure 8-2 shows what happens when a modification is made to the rationale.

Figure 8-3 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 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 (lower 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 dissertation come from a conference room 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 software system that the rationale describes is being developed. SEURAT supports the entry by providing rationale entry screens for each type of rationale element.

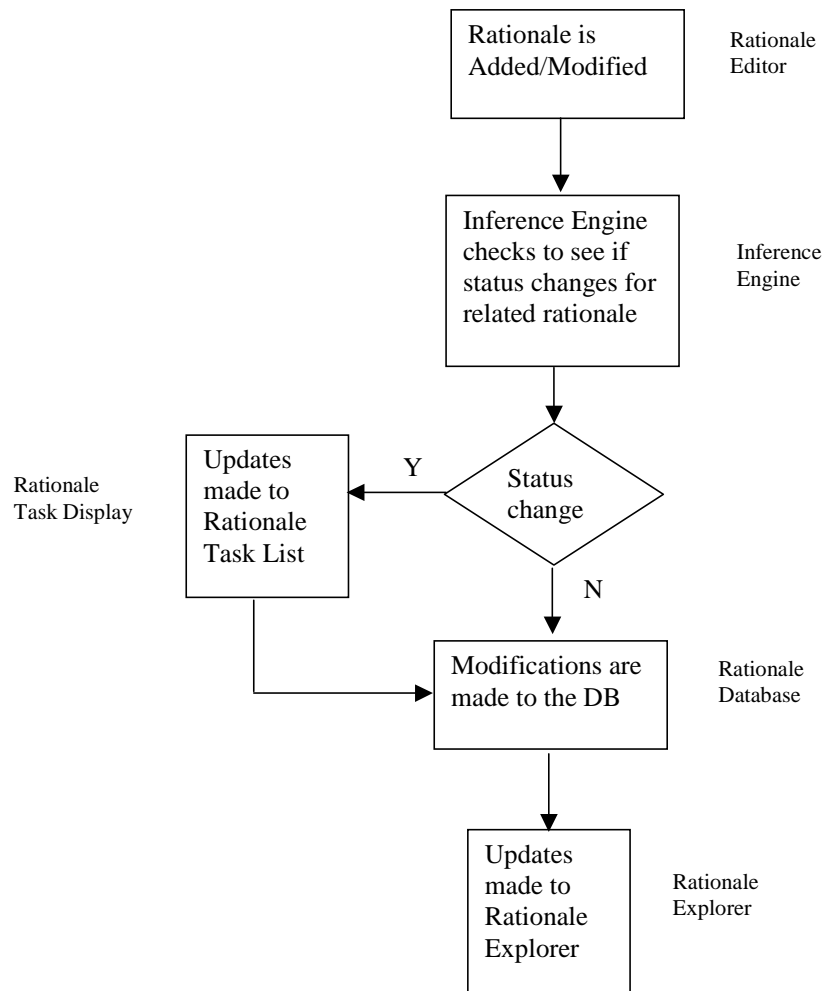


FIGURE 8-2. Rationale Update Flowchart

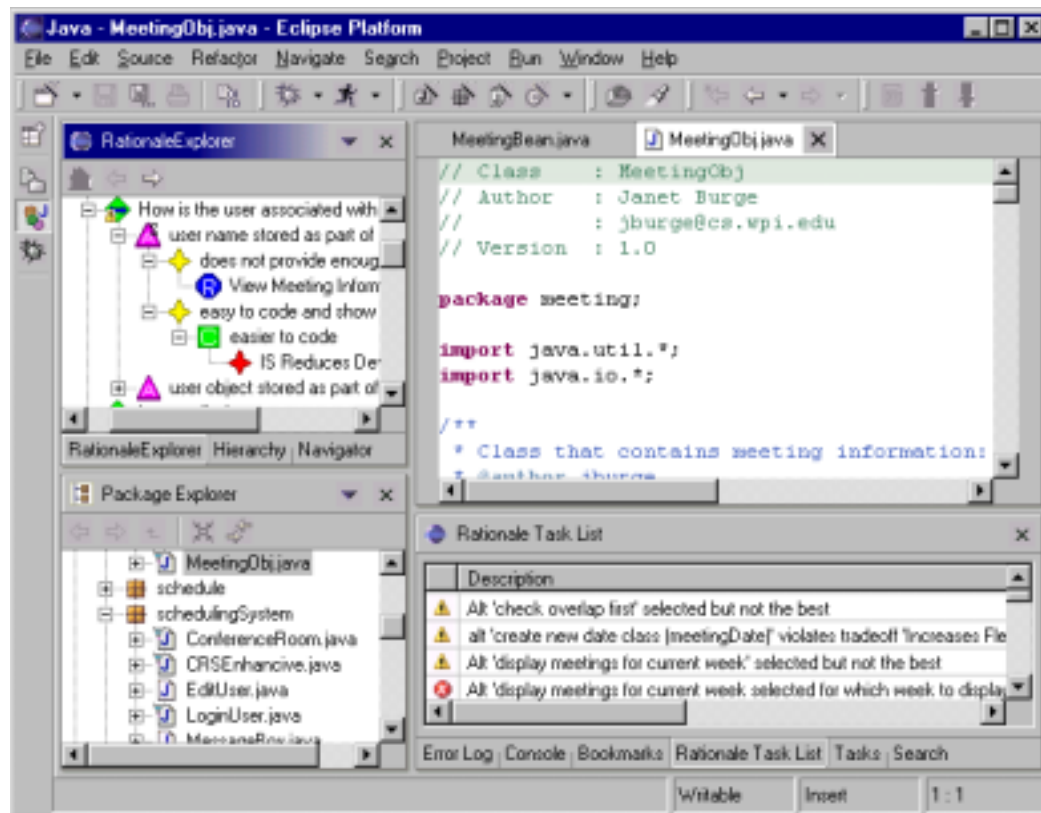


FIGURE 8-3. SEURAT and Eclipse

8.2. Rationale Repository and Argument Ontology

The Rationale Repository and Argument Ontology are all stored in a relational database. The database used for this was MySQL (<http://www.mysql.com/>). The repository was created by translating the information stored in the XML format described in the previous chapter into a series of database tables. These consist of tables for each rationale element and, in some cases, tables giving the relationships between the rationale elements. Using the relational database allows the inferences to be performed by taking advantage of the power of SQL queries. The database also provides a more scalable solution than flat files (such as XML) or facts in a rule base because the rationale does not all need to be in memory at one time. The following sections describe what is stored, and how.

8.2.1. Requirements

Figure 8-4 gives the SQL table definition for a Requirement. This includes the following fields:

-
- *id* – this provides the unique identifier for the requirement;
 - *name* – this provides a short description that is used in the Rationale Explorer display;
 - *description* – this gives a longer description presenting the text of the requirement;
 - *type* – indicates if this is a functional or non-functional requirement;
 - *status* – the status of the requirement;
 - *artifact* – an artifact identifier that can be used to map the requirement to a requirements document or repository;
 - *ptype* – indicates the type of “parent” for this requirement. This allows a requirement to be a sub-element of another requirement;
 - *parent* – the identifier of the parent;
 - *enabled* – indicates if this is an active requirement. Requirements can be disabled if they have been deferred or removed.

```
CREATE TABLE Requirements (
    id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY
                KEY,
    name        VARCHAR(80),
    description  VARCHAR(120),
    type        ENUM("FR", "NFR"),
    status       ENUM("Satisfied", "Violated", "Addressed",
                    "Retracted", "Rejected", "Undecided",
                    "Deferred"),
    artifact     VARCHAR(80),
    ptype       ENUM("None", "Requirement"),
    parent      INT UNSIGNED,
    enabled      ENUM ("False", "True"),
    index(name),
    index(ptype, parent));
```

FIGURE 8-4. Requirement SQL Definition

8.2.2. Decision

Figure 8-5 gives the SQL table definition for a Decision. This includes the following fields:

- *id* – this provides the unique identifier for the decision;
- *name* – this provides a short description that is used in the Rationale Explorer display;

-
- *description* – this provides a longer description with a more detailed description of the decision;
 - *type* – indicates if this is a single choice (only one alternative can be selected) or a multiple choice decision;
 - *status* – the status of the decision;
 - *phase* – the phase of development during which the decision is/was made;
 - *ptype* – indicates the type of “parent” for this decision. This can be a requirement (the decision had to be made because of the requirement), another decision (this decision is a sub-decision of a higher-level one), or an alternative (choosing the alternative meant that the decision needed to be made);
 - *parent* – the identifier of the parent.

```
CREATE TABLE Decisions (
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY
              KEY,
  name        VARCHAR(80),
  description VARCHAR(120),
  type        ENUM("SingleChoice", "MultipleChoice"),
  status      ENUM("Resolved", "Unresolved",
                  "Non-resolvable", "Retracted",
                  "Addressed"),
  phase       ENUM("Requirements", "Analysis", "Design",
                  "Implementation", "Test", "Maintenance"),
  subdecreq   ENUM("Yes", "No"),
  ptype       ENUM("None", "Requirement", "Decision",
                  "Alternative"),
  parent      INT UNSIGNED,
  index (name),
  index (ptype, parent));
```

FIGURE 8-5. Decision SQL Definition

8.2.3. Alternative

Figure 8-6 gives the SQL table definition for an Alternative. This includes the following fields:

- *id* – this provides the unique identifier for the alternative;
 - *name* – this provides a short description that is used in the Rationale Explorer display;
-

- *description* – this provides a longer description with more details about the alternative;
- *status* – the status of the alternative;
- *artifact* – an artifact identifier that can be used to map the alternative to the associated code or other development artifact(s);
- *ptype* – indicates the type of “parent” for this alternative. This is usually a decision;
- *parent* – the identifier of the parent.

```
CREATE TABLE Alternatives (
    id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY
                KEY,
    name        VARCHAR(80),
    description  VARCHAR(120),
    status      ENUM("Valid", "Adopted", "At_Issue",
                    "Rejected", "Retracted"),
    evaluation  FLOAT,
    ptype       ENUM("None", "Decision"),
    parent      INT UNSIGNED,
    index(name),
    index(ptype, parent));
```

FIGURE 8-6. Alternative SQL Definition

8.2.4. Argument

Figure 8-7 gives the SQL table definition for an Argument. This is the most complicated of the rationale items because it gives the information needed to evaluate the strength of the various alternatives. The Argument table includes the following fields:

- *id* – this provides the unique identifier for the argument;
- *name* – this provides a short description that is used in the Rationale Explorer display;
- *description* – this provides a longer description with more details about the argument;
- *ptype* – indicates the type of “parent” for this argument. This can be a Requirement, Decision, Alternative, or another Argument;
- *parent* – the identifier of the parent;
- *type* – this is the type of argument. The values vary depending on what type of argument it is. For arguments about requirements, valid values are Addresses, Satisfies, and Violates. For claims, valid values are Denies and Supports. For other alternatives, valid values are Pre-supposes, Pre-supposed-by, Opposed, and Opposed-by;

- *plausibility* – provides the degree of confidence the developer has in the argument;
- *importance* – provides the importance of the argument. This can be inherited from a claim by specifying it as Default;
- *amount* – provides the strength of the argument. For example, if an alternative is cheap, how cheap?
- *argtype* – provides the type of item the argument is about. This can be a Claim, Alternative, Requirement, or Assumption;
- *claim*, *alternative*, *requirement*, *assumption* – only one of these fields has a non-null value that gives the id of the element to which it maps.

```
CREATE TABLE Arguments (
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY
              KEY,
  name        VARCHAR(80),
  description VARCHAR(120),
  ptype       ENUM("Requirement", "Decision", "Alternative",
                  "Argument"),
  parent      INT UNSIGNED DEFAULT NULL,
  type        ENUM("Denies", "Supports", "Pre-supposes",
                  "Pre-supposed-by", "Opposed",
                  "Opposed-by", "Addresses", "Satisfies",
                  "Violates"),
  plausibility ENUM("Low", "Medium", "High", "Certain"),
  importance   ENUM("Default", "Not", "Low", "Moderate",
                  "High", "Essential"),
  amount       INT UNSIGNED,
  argtype      ENUM("Claim", "Alternative", "Requirement",
                  "Assumption"),
  claim        INT UNSIGNED DEFAULT NULL,
  alternative  INT UNSIGNED DEFAULT NULL,
  requirement  INT UNSIGNED DEFAULT NULL,
  assumption   INT UNSIGNED DEFAULT NULL,
  index(ptype, parent));
```

FIGURE 8-7. Argument SQL Definition

8.2.5. Claim

Figure 8-8 gives the SQL table definition for a Claim. Claims map to elements in the Argument Ontology but provide additional information by adding a qualifier of “IS” or “NOT”. This specifies the direction of the claim – it indicates if the argument is saying that the alternative has the quality or is in violation of it. For example, this would let you specifically state that an argument says “NOT reduces coupling,” i.e. this choice causes coupling. The Claim table includes the following fields:

- *id* – this provides the unique identifier for the claim;
- *name* – this provides a short description that is used in the Rationale Explorer display;
- *description* – this provides a longer description with more details about the claim;
- *direction* – states if the claim is that an alternative “IS” or “IS NOT” a possessor of the claimed property;
- *importance* – provides the importance of the claim. This can be inherited from an Ontology Entry by specifying it as Default;
- *ontology* – provides the id of the corresponding Ontology Entry;
- *enabled* – states if this claim is enabled or disabled.

```
CREATE TABLE Claims (  
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY  
              KEY,  
  name        VARCHAR(80),  
  description  VARCHAR(120),  
  direction   ENUM("IS", "NOT"),  
  importance  ENUM("Default", "Not", "Low", "Moderate",  
                  "High", "Essential"),  
  ontology    INT UNSIGNED,  
  enabled     ENUM ("False", "True"),  
  index(name),  
  index(id));
```

FIGURE 8-8. Claim SQL Definition

8.2.6. Assumption

Figure 8-9 gives the SQL table definition for an Assumption. Assumptions are things that the developer believes to be true at the time the software is being developed and which may change in the future. The Assumption table includes the following fields:

-
- *id* – this gives the unique identifier for the assumption;
 - *name* – this provides a short description that is used in the Rationale Explorer display;
 - *description* – this provides a longer description with more details about the assumption;
 - *enabled* – states if this assumption is enabled or disabled.

```
CREATE TABLE Assumptions (  
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY  
              KEY,  
  name        VARCHAR(80),  
  description VARCHAR(120),  
  importance  ENUM("Default", "Not", "Low", "Moderate",  
                  "High", "Essential"),  
  enabled     ENUM ("False", "True"),  
  index(name),  
  index(id));
```

FIGURE 8-9. Assumption SQL Definition

8.2.7. Ontology Entry

Figure 8-10 gives the SQL table definition for an Ontology Entry. The Ontology Entry table includes the following fields:

- *id* – this provides the unique identifier for the ontology entry;
- *name* – this provides a short description that is used in the Rationale Explorer display;
- *description* – this provides a longer description with more details about the ontology entry;
- *importance* – provides the importance of the ontology entry.

The Ontology Entries are stored in a hierarchy that forms the Argument Ontology. A separate table is used to store the parent-child relationships between entries. Figure 8-11 shows that table.

```
CREATE TABLE OntEntries (  
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY  
              KEY,  
  name        VARCHAR(80),  
  description  VARCHAR(120),  
  importance   ENUM("Not", "Low", "Moderate", "High",  
                    "Essential"),  
  index(name),  
  index(id));
```

FIGURE 8-10. Ontology Entry SQL Definition

```
CREATE TABLE OntRelationships (  
  parent INT UNSIGNED,  
  child INT UNSIGNED);
```

FIGURE 8-11. Ontology Relationships SQL Definition

8.2.8. Tradeoffs

Figure 8-12 gives the SQL table definition for Tradeoffs. Tradeoffs come in two types: Tradeoffs, which give two ontology entries that are typically traded off against each other, and Co-occurrences, two ontology entries that usually occur together on the same side of an argument. The Tradeoff table includes the following fields:

- *id* – this provides the unique identifier for the tradeoff;
- *name* – this provides a short description that is used in the Rationale Explorer display;
- *description* – this provides a longer description with more details about the tradeoff;
- *ontology1* – the first Ontology Entry traded off;
- *ontology2* – the second Ontology Entry traded off;
- *type* – specifies if this is a tradeoff or co-occurrence;
- *symmetric* – indicates if the tradeoff is symmetric. For example, more flexible solutions may cost more than cheaper ones but just because a choice is more expensive does not mean it is more flexible.

```
CREATE TABLE Tradeoffs (
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY
              KEY,
  name        VARCHAR(80),
  description  VARCHAR(120),
  ontology1   INT UNSIGNED,
  ontology2   INT UNSIGNED,
  type        ENUM("Tradeoff", "Co-Occurrence"),
  symmetric   ENUM("False", "True"));
```

FIGURE 8-12. Tradeoff SQL Definition

8.2.9. Questions

Figure 8-13 gives the SQL table definition for Questions. Questions can be about Requirements, Decisions, or Alternatives and refer to information that is needed to help make decisions. The Question table includes the following fields:

- *id* – this provides the unique identifier for the tradeoff;
- *name* – this provides a short description that is used in the Rationale Explorer display;
- *description* – this provides a more detailed statement of the question;
- *status* – indicates if the question is answered or not;
- *proc* – provides the recommended procedure for finding the answer. This could specify who to ask or if there are experiments that need to be run;
- *answer* – the answer to the question;
- *ptype* – specifies if this question is about a Requirement, Decision, or Alternative.

8.2.10. History

Figure 8-14 shows the history that is stored for each element that can change state based on user action. These elements are Requirements, Decisions, Alternatives, and Questions. The History table includes the following fields:

- *ptype* – specifies whether this history is for a Requirement, Decision, Alternative, or Question;
 - *parent* – the id of the Requirement, Decision, Alternative, or Question that the history is for;
 - *date* – the date the history element was recorded;
 - *reason* – the reason the state of the item changed;
-

-
- *status* – the new status of the item after the change.

```
CREATE TABLE Questions (  
  id          INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY  
              KEY,  
  name        VARCHAR(80),  
  description  VARCHAR(120),  
  status      ENUM("Unanswered", "Answered"),  
  proc        VARCHAR(80),  
  answer      VARCHAR(80),  
  ptype       ENUM("None", "Requirement", "Decision",  
                  "Alternative", "Argument"),  
  parent      INT UNSIGNED DEFAULT NULL,  
  index(name),  
  index(ptype, parent));
```

FIGURE 8-13. Question SQL Definition

```
CREATE TABLE History (  
  ptype       ENUM("Requirement", "Decision", "Alternative",  
                  "Question"),  
  parent      INT UNSIGNED,  
  date        TIMESTAMP,  
  reason      VARCHAR(80),  
  status      VARCHAR(20),  
  index(ptype, parent));
```

FIGURE 8-14. History SQL Definition

8.2.11. Status

When errors are detected in the rationale, they are stored in the Status table shown in Figure 8-15. This table contains the following fields:

- *ptype* – specifies whether the status is for a Requirement, Decision, or Alternative
- *parent* – the id of the Requirement, Decision, or Alternative that the status is for
- *date* – the date the status element was recorded
- *type* – the error level

-
- *description* – a description of the problem
 - *status* – a status identifier stored as a string (this allows additional status types to be added without changing the table definition)
 - *override* – a flag indicating if the display of this status item has been overridden by the user

```
CREATE TABLE Status (  
    parent      INT UNSIGNED,  
    ptype       ENUM("Requirement", "Decision",  
                    "Alternative"),  
    date        TIMESTAMP,  
    type        ENUM("Information", "Warning", "Error"),  
    description  VARCHAR(255),  
    status      VARCHAR(40),  
    override     ENUM("Yes", "No"),  
    index(ptype, parent, status));
```

FIGURE 8-15. Status SQL Definition

8.2.12. Associations

The Associations table, shown in Figure 8-16, gives information about the associations between the code and the alternatives. This table contains the following fields:

- *alternative* – gives the id for the alternative being associated;
- *artifact* – specifies the artifact associated with the alternative. This is a fully qualified specification in the format needed by Eclipse;
- *artResource* – specifies the name of the code file;
- *artName* – specifies the name of the resource – this could be a file, class, method, or attribute name;
- *assocMessage* – specifies the message displayed in the bookmark that describes the association;
- *lineNumber* – gives the line in the code that is associated with the rationale.

```
CREATE TABLE Associations (  
    alternative INT UNSIGNED,  
    artifact    VARCHAR(255),  
    artResource VARCHAR(120),  
    artName     VARCHAR(120),  
    assocMessage VARCHAR(255),  
    lineNumber  INT UNSIGNED);
```

FIGURE 8-16. Association SQL Definition

8.3. Rationale Explorer

The user's main access to the rationale is through the Rationale Explorer. The Rationale Explorer is modeled after the Java Package Explorer used in Eclipse. The rationale is read in from the MySQL database when SEURAT is started up. Figure 8-17 shows the top level of rationale stored in the Rationale Explorer.

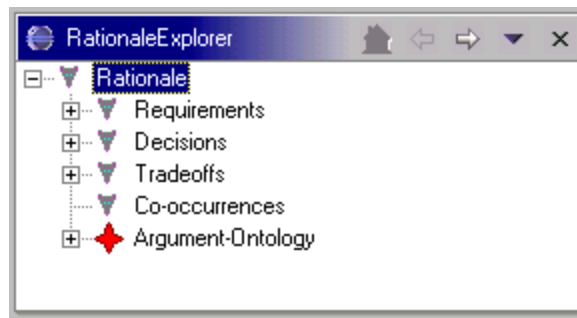


FIGURE 8-17. Rationale Explorer – Top Level Rationale

The rationale elements can be expanded to show the rationale in a hierarchical form. Figure 8-18 shows the Rationale Explorer with parts of the tree expanded to show the details of the rationale.

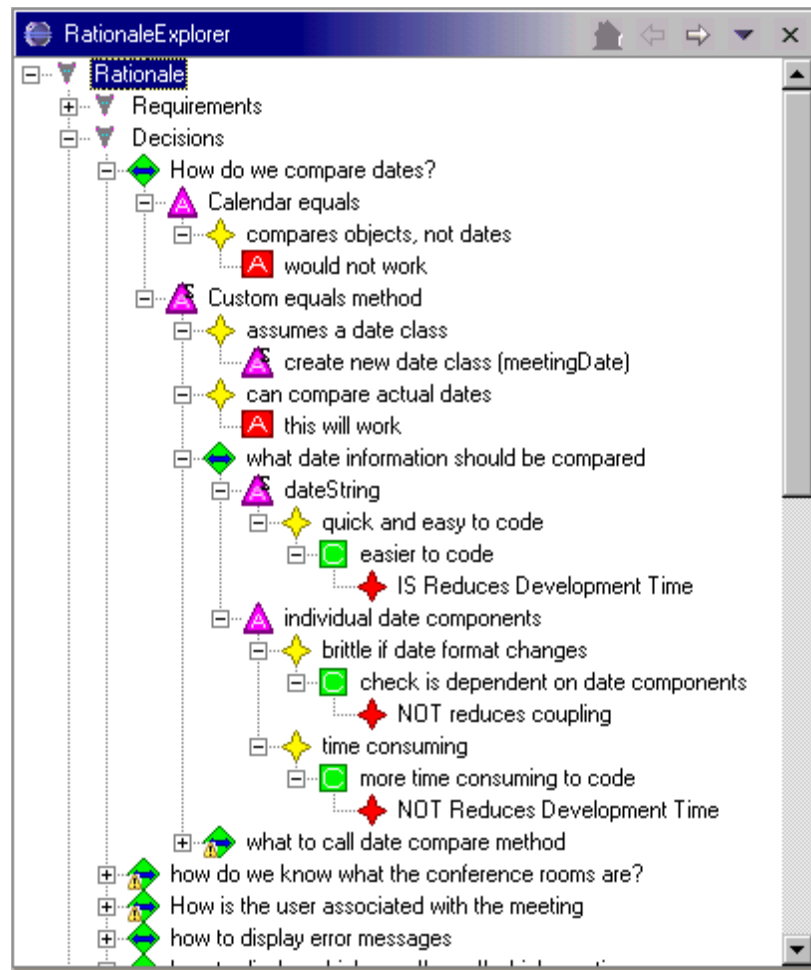


FIGURE 8-18. Rationale Explorer with Expanded Rationale

Each type of rationale element has a different icon describing it. These icons are listed in Figure 8-19. In addition, the icons can have smaller icons overlaid to indicate the status of the rationale item. There are several examples shown in Figure 8-18. The alternative “Custom equals method” has a small “S” overlaid on the upper right side of the purple triangle icon to indicate that the alternative has been selected. The decision “what to call date compare method” has a yellow triangle with an exclamation point in it overlaid on the lower left side of the green diamond that indicates that there is a warning. Other overlays include a “D” for disabled, which would be shown in the upper right of the icon, and a red square with a white “X” in it, which would be shown in the lower left of the icon and would indicate that the element had an error. Figure 8-20 shows examples of each type of overlay.











	Requirement
	Decision
	Alternative
	Argument
	Claim
	Assumption
	Question
	Tradeoff
	Co-occurrence
	Ontology Entry

FIGURE 8-19. Rationale Element Icons





	Requirement with Error
	Decision with Warning
	Selected Alternative
	Disabled Assumption

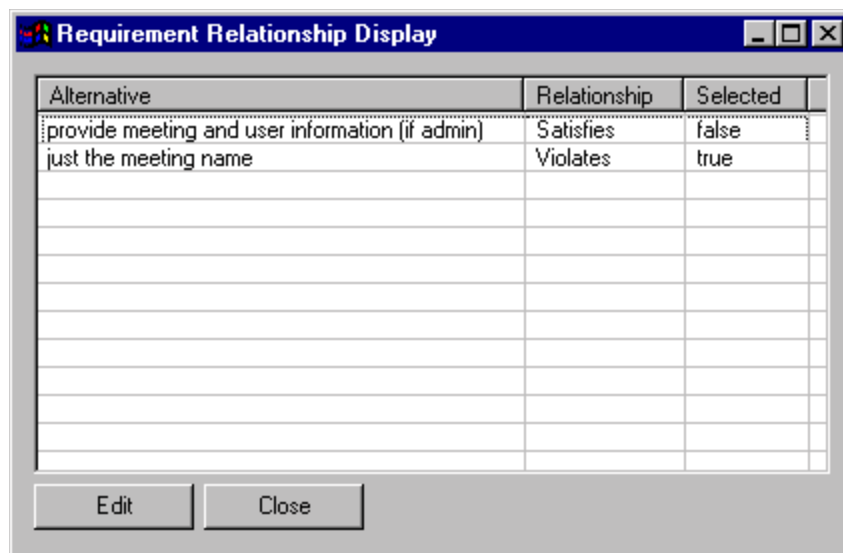
FIGURE 8-20. Icon Overlay Examples

The Rationale Explorer gives context sensitive menus for each of the rationale elements that can be displayed by right-clicking on the element. These menus let the user perform various actions. The following sections describe the items on each menu.

8.3.1. Requirement Menu

The top-level requirement gives only one menu item: New. This is a way to enter requirements into the system. Each individual requirement has the following items in its menu:

- *Edit* – brings up an editor for the requirement. This can also be done by double-clicking the requirement.
- *Delete* – deletes the requirement. This is only allowed if there are no other references to the requirement in the rationale (such as arguments that refer to it).
- *New Argument* – creates a new argument for or against the requirement.
- *Find Relationships* – looks for alternatives that are related to the requirement. Figure 8-21 shows an example of this display. This shows how the alternative is related to the requirement and whether the alternative is selected or not.
- *Show History* – shows a list of status changes, and the reasons behind them, for the requirement.

A screenshot of a software window titled "Requirement Relationship Display". It contains a table with three columns: "Alternative", "Relationship", and "Selected". The table has two rows of data. The first row shows "provide meeting and user information (if admin)" as the alternative, "Satisfies" as the relationship, and "false" as the selected status. The second row shows "just the meeting name" as the alternative, "Violates" as the relationship, and "true" as the selected status. Below the table are two buttons: "Edit" and "Close".

Alternative	Relationship	Selected
provide meeting and user information (if admin)	Satisfies	false
just the meeting name	Violates	true

FIGURE 8-21. Requirement Relationship Display

8.3.2. Decision Menu

The top-level decision gives only one menu item: New. This is a way to enter decisions into the system. Each individual decision has the following items in its menu:

-
- *Edit* – brings up an editor for the decision. This can also be done by double-clicking the decision.
 - *Delete* – deletes the decision. This is only allowed if there are no sub-elements in the rationale that refer to this decision.
 - *New Alternative* – creates a new alternative for the decision.
 - *New Question* – creates a new question about the decision.
 - *Show History* – shows a list of status changes and the reasons behind them for the decision.

8.3.3. Alternative Menu

Each alternative has the following items in its menu:

- *Edit* – brings up an editor for the alternative. This can also be done by double-clicking the alternative.
- *Delete* – deletes the alternative. This is only allowed if there are no sub-elements in the rationale that refer to this alternative.
- *Associate* – associates the alternative with an element of code selected using the Java Package Explorer. This will bring up a dialog box confirming the association.
- *New Argument* – creates a new argument for the alternative.
- *New Question* – creates a new question about the alternative.
- *New Decision* – creates a new decision required when, or if, the alternative is chosen.
- *Show History* – shows a list of status changes and the reasons behind them for the alternative.

8.3.4. Argument Menu

The Argument Menu has only two elements: Edit and Delete. Editing can also be performed by double-clicking on the argument. If an argument is deleted, any Claims and Assumptions associated with it will be deleted also unless they are referred to by more than one argument. This is done to avoid cluttering the rationale database with Claims and Assumptions that are no longer used.

8.3.5. Claim Menu

The Claim Menu only has an Edit option. A claim can only be deleted by removing the associated Argument.

8.3.6. Assumption Menu

Assumptions, like claims, only have an Edit option.

8.3.7. Question Menu

Each alternative has the following items in its menu:

- *Edit* – brings up an editor for the question. This can also be done by double-clicking the question.
- *Delete* – deletes the question.
- *Show History* – shows a list of status changes and the reasons behind them for the question.

8.3.8. Tradeoff and Co-Occurrence Menu

The top-level tradeoff and co-occurrence gives only one menu item: New. This is a way to enter new tradeoffs and co-occurrences into the system. Each individual item has the following items in its menu:

- *Edit* – brings up an editor for the tradeoff/co-occurrence. This can also be done by double-clicking the element.
- *Delete* – deletes the tradeoff/co-occurrence.

8.3.9. Ontology Entry Menu

The top-level ontology entry gives only one menu item: New. This is a way to enter higher-level entries into the system. Each individual item has the following items in its menu:

- *Edit* – brings up an editor for the ontology entry. This can also be done by double-clicking the element.
- *New* – adds a new ontology entry. This lets the user select an entry from the tree or create a new one. This is done because an entry is allowed to be in multiple places in the tree.
- *Delete* – deletes the ontology entry. This will only be allowed if the entry has no sub-elements.

8.4. Inference Engine

As described in the previous chapter, there are many inferences that can be performed over the rationale to check it for completeness and consistency. The following sections describe what the possible errors are, how they are displayed, and how inference is performed by SEURAT.

8.4.1. Error and Warning Visualization

Any errors or warnings computed by the inferences are displayed in two ways. One way is by placing an error or warning icon on the icon for the rationale element. These are always shown over the lower left hand corner of the element. The second way is on the Rationale Task Display. This display is modeled after the Task List used to display Java compilation warnings and errors. Figure 8-22 shows the SEURAT Rationale Task List.

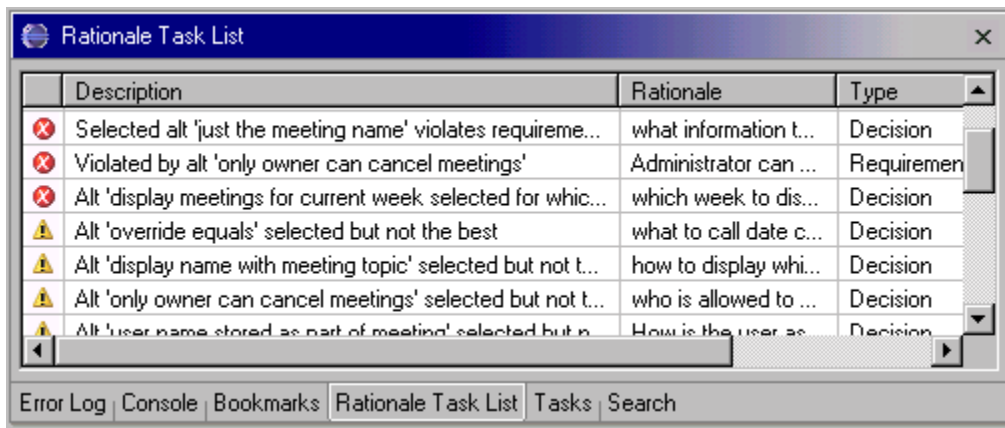


FIGURE 8-22. Rationale Task List

There are a number of different errors and warnings that can be displayed by SEURAT. Table 8-1 presents a table showing the error text displayed, the code identifying the error type, and the type of rationale element that it will be displayed for. The error level is indicated on the display by the type of icon used – a yellow triangle with an exclamation point for warnings and a red square with an “X” in it for errors.

TABLE 8-1. Rationale Task Messages

Error Text	Error Type	Element Type	Error Level
No alternative selected	NONE_SELECTED	Decision	Error
Sub-decisions are required but missing	SUBDECISIONS_MISSING	Decision	Error
Multiple alternatives selected for the decision	MULTIPLE_SELECTION	Decision	Error
Alt <name> selected but not the best	LESS_SUPPORTED	Decision	Warning
Alt <name> selected for <name> has arguments against it but none for it	SELECTED_ONLY_AGAINST	Decision	Error
Alt <name> selected for <name> has no arguments in its favor	SELECTED_NONE_FOR	Decision	Warning
Selected alt <name> violates requirement <name>	ALT_REQ_VIOLATION	Decision	Error
Alt <name> requires non-selected alt <name>	PRESUPPOSED_NOTSEL	Decision	Error
Alt <name> is opposed by selected alt <name>	OPPOSED_SEL	Decision	Error
Question <name> about selected alt is not answered	UNANSWERED_ALT_QUESTION	Decision	Warning
Question <name> is unanswered	UNANSWERED_QUESTION	Decision	Warning
Alt <name> violates tradeoff <name> vs. <name>: tradeoff is contradicted	TRADE_VIOLATION	Decision	Error
Alt <name> violates tradeoff <name> vs. <name>: second element is missing	TRADE_VIOLATION	Decision	Warning

Alt <name> violates tradeoff <name> vs. <name>: first element is missing	TRADE_VIOLATION	Decision	Warning
Alt <name> violates co-occurrence <name> vs. <name>: co-occurrence is contradicted	CO_OCCURRENCE_VIOLATION	Decision	Error
Alt <name> violates co-occurrence <name> vs. <name>: second element is missing	CO_OCCURRENCE_VIOLATION	Decision	Warning
Alt <name> violates co-occurrence <name> vs. <name>: first element is missing	CO_OCCURRENCE_VIOLATION	Decision	Warning
Question <name> about alt is unanswered	UNANSWERED_ALT_QUERY	Alternative	Warning
The same, or opposite argument, both supports and opposes alt <name>	CONTRADICTIONARY_ARGUMENTS	Alternative	Error
Contradictory arguments appear on the same side for alt <name>	CONTRADICTIONARY_ARGUMENTS	Alternative	Error
Duplicate arguments found regarding <name>	DUPLICATE_ARGUMENTS	Alternative	Warning
Violated by alt <name>	REQ_VIOLATION	Requirement	Error

8.4.2. Error and Warning Detection

The inferences that detect the errors and warnings were initially encoded as CLIPS rules but the final implementation used a combination of SQL queries and Java code. Whenever an element is modified, created, or deleted, an `UpdateStatus` method is called that will return a list of `RationaleStatus` objects that are then used to update the status displayed in the Rationale Explorer and on the Rationale Task List for that element, as well as on any elements related to it. Each `UpdateStatus` method gets the element being tested from the database, along with any other elements it is the parent of, and checks to see if the status has changed. The list of `RationaleStatus` objects returned from the evaluation is compared to the current status of the element so that if problems with the rationale that caused errors

or warnings earlier have been corrected, those errors and warnings can be removed. The following sections describe the checks made when the elements are modified.

8.4.2.1. Requirement Inferences

When a requirement is added or modified, and the requirement is “active” (i.e., not disabled or deferred), the status of the requirement is computed. The first check is to see if there are any selected alternatives that have arguments stating that they violate the requirement. This is checked by using an SQL query to get the arguments that describe the violation and then retrieving the alternatives referred to in those arguments. If the alternatives are selected, the violation is reported as an error.

If the requirement has been disabled or deferred, inferencing is performed over any alternatives that refer to it. This is done to re-evaluate the alternative without counting arguments that refer to the no longer active requirement.

8.4.2.2. Decision Inferences

When a decision is added or modified, the status of that decision is computed. The decision, and all rationale elements that refer to it (its alternatives and questions) are retrieved from the database. The following checks are performed using Java:

- *Selected alternatives* – if the decision is not broken into sub-decisions, it should have a selected alternative. If there is no selected alternative, that is reported as an error. If there are more than one selected alternative, and the decision is a single-choice decision, that is also reported as an error.
- *Sub-decisions* – if the decision is supposed to be broken into sub-decisions, SEURAT checks to make sure that the sub-decisions exist. If not, that is reported as an error.
- *Arguments in favor* – the arguments for the selected alternative are examined to see if any of them are in favor of that alternative. A warning is reported if there are no arguments referring to the alternative. An error is reported if there are no arguments in its favor but are arguments against it.
- *Requirements violations* – the arguments for the selected requirement are checked to see if any report requirement violations. If so, the violation is reported as an error and inferences are performed on the requirement so that its status is updated also. This does result in requirement violations reported in two different ways but this is not considered a flaw because the seriousness merits the extra information given by reporting the violation twice.
- *Alternative relationships* – SEURAT checks to see if there are any arguments that pre-suppose or oppose other alternatives. If pre-supposed alternatives are not selected, this is reported as an error. If opposed alternatives are selected, this is reported as an error.

The reverse check is performed if an alternative is edited (see the section on Alternative Inferences).

- *Question status* – if there are any questions associated with this decision that have not been answered that is reported as an error. This is also performed for any questions associated with the alternatives for the decision.
- *Tradeoff violations* – all tradeoffs stored in the rationale are checked to make sure that the alternative selected for this decision does not violate any of them. This is done using a combination of SQL queries (to find the applicable tradeoffs) and Java code.
- *Decision evaluation* – each argument is evaluated to determine how well it is supported. This was done using the following formula, based on the evaluations originally done by our earlier prototype, InfoRat:

$$AltEvaluation = \sum_{arg-for=i} amt_i * imp_i - \sum_{arg-against=j} amt_j * imp_j$$

This calculation requires that three things be known about each argument: the amount, the importance, and whether the argument is for or against the alternative. The amount is entered by the user when they record the argument. The other two characteristics are computed based on values stored in the rationale and on the type of rationale element that is used in the argument.

For arguments concerning requirements, arguments that indicate that the alternative violates a requirement are considered as being against the alternative, while arguments that state the alternative addresses or satisfies the requirement are considered to be for the alternative. The importance of an argument that is about a requirement is always given the maximum importance (Essential), which translates to the number one.

For arguments concerning claims, arguments that indicate that the alternative denies the claim are considered to be against the alternative, while arguments that state the alternative supports the claim are considered as being for the alternative. The importance of an argument is either taken directly from the argument itself (as entered by the user) or, if the importance is the default, it is inherited from the claim or the ontology entry the claim describes.

For arguments concerning assumptions, arguments that indicate that the alternative denies the assumption are considered to be against the alternative, while arguments that state the alternative supports the assumption are considered to be for the alternative. The importance of an argument is taken directly from the argument itself (as entered by the user).

For arguments concerning other alternatives, the calculation is a bit more difficult. These express dependencies between different alternatives and the role that the dependency plays in the evaluation will depend on whether the other alternative is selected or not. If the argument states that the alternative being evaluated opposes another alternative, and that other alternative is selected, the argument will be given the maximum importance and will be counted as an argument against the alternative being evaluated. If the argument states that the alternative being evaluated pre-supposes another alternative, and that other alternative is not selected, the argument will be given the maximum importance and will be counted as an argument against the alternative being evaluated.

8.4.2.3. Alternative Inferences

When an alternative is added or modified, the status of that alternative is computed. The decision, and all rationale elements that refer to it (its arguments and questions) are retrieved from the database. The following checks are performed using Java:

- *Question status* – if there are any unanswered questions about this alternative an error is reported.
- *Alternative relationships* – SEURAT checks to see if there are any arguments for other alternatives that state that they pre-suppose or oppose this alternative. another alternative pre-supposes this one, and this one is not selected, an error is reported. If another alternative opposes this one, and this one is selected, an error is reported.
- *Contradictory arguments* – if there is an argument for the alternative that is the same as an argument against the alternative, the contradiction is reported as an error. This includes arguments that refer to claims, assumptions, and requirements.
- *Duplicate arguments* – if there are two arguments for (or against) the alternative that refer to the same claim, assumption, or requirement, that is reported as an error.
- *Requirement inferences* – if this alternative is not selected, inferencing is performed over any requirements associated with it (via the arguments) to clear any violations that may have been reported earlier.
- *Decision inferences* – the inferencing over the decision is repeated to account for any changes in the status of the alternative (see the section on Decision Inferences).

8.4.2.4. Question Inferences

If a question is added, edited, or deleted, inferencing is performed over the decision or alternative that it refers to.

8.4.2.5. Argument Inferences

If an argument is added, edited, or deleted, inferencing is performed over the alternative or requirement that refer to it.

8.4.2.6. Claim Inferences

If a claim is added or edited, inferencing is performed over any arguments that refer to it (claims can be referenced by multiple arguments).

8.4.2.7. Assumption Inferences

If an assumption is added or edited, inferencing is performed over any arguments that refer to it (claims can be referenced by multiple arguments).

8.4.2.8. Ontology Inferences

If an element of the argument ontology is added or edited, inferencing is performed over any claims that refer to it.

8.4.2.9. Tradeoff Inferences

If a tradeoff is added, edited, or deleted, inferencing is performed over all decisions to insure that the tradeoff is not violated.

8.5. Rationale to Code Associations

The rationale provides useful insight into the reasons behind design and implementation choices. This is only useful, however, if the maintainer is aware that the rationale is there and is able to easily view it. One of the key features in SEURAT is the ability to make the rationale to code associations explicit and to support the ability to go from rationale to code and vice versa.

SEURAT allows the developer to associate each alternative with the code that implements it. This can be done at the file, class, method, or attribute level. SEURAT shows that the associations are present in several ways:

- “decorations” on the icons in the Package Explorer to show which files have associated rationale;
- artifact names given in the alternative descriptions;
- bookmark icons inside the editor next to the affected code that lists the alternative(s) when the mouse is placed over it;

-
- bookmarks giving the alternative and the associated code artifact that will bring up the associated code when the maintainer double-clicks them.

The maintainer associates code with rationale by selecting the code in the Package Explorer and then choosing “Associate” by right-clicking on the alternative. This will display the name of the selected code item so the user can verify that this is the association they want. The icon next to the class that contains the code will then be marked with a small “rat” icon. Figure 8-23 shows the Package Explorer where some of the classes (MeetingDate and MeetingObj) have rationale associated with them. This is denoted by a small “rat” icon that is overlaid on the upper left-hand corner of the file icon. An example icon is shown in Figure 8-24.

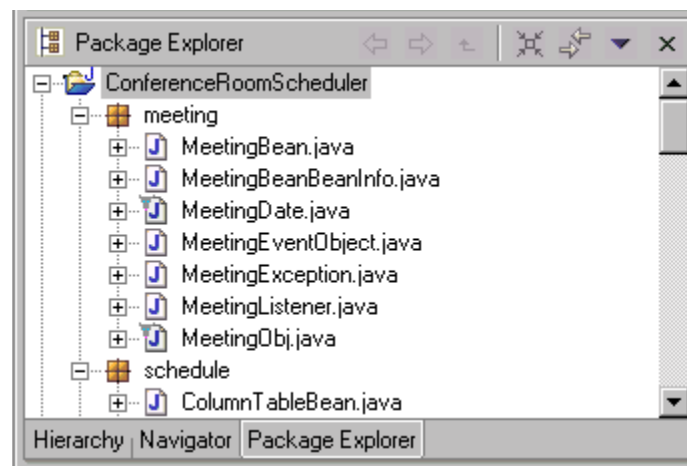


FIGURE 8-23. Package Explorer with Associations

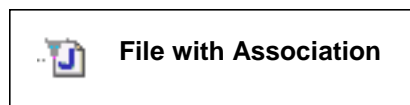


FIGURE 8-24. Association Icon

When the code is associated with the rationale, the association is stored in the Rationale Repository (as shown in the Associations table in Section Associations) and a “Bookmark” is added to the Eclipse Bookmark View. This shows the name of the

alternative, the file it is associated with, the folder the file can be found in, and the line number in the file. Figure 8-25 shows the Bookmark View.

The user can also look for the alternative in the Bookmark View and, by double-clicking the bookmark entry, can bring up the associated code in the editor.

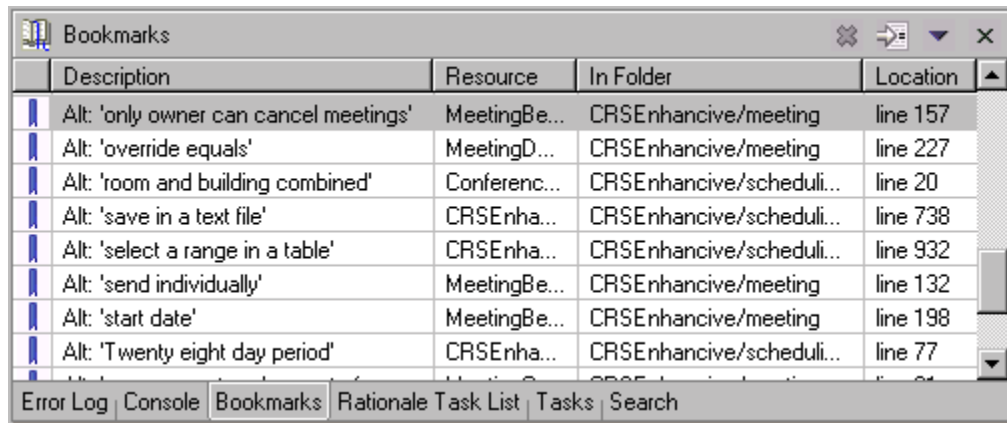


FIGURE 8-25. Bookmark View

Alternative Information

Name: Conference room class

Description: Create a special class to contain the conference room information

Status: Adopted

Artifact: ConferenceRoom.java

Arguments For
can be extended to hold more information
will hold the required information

Arguments Against

Relationships

Save Cancel

FIGURE 8-26. Alternative Showing Code Association

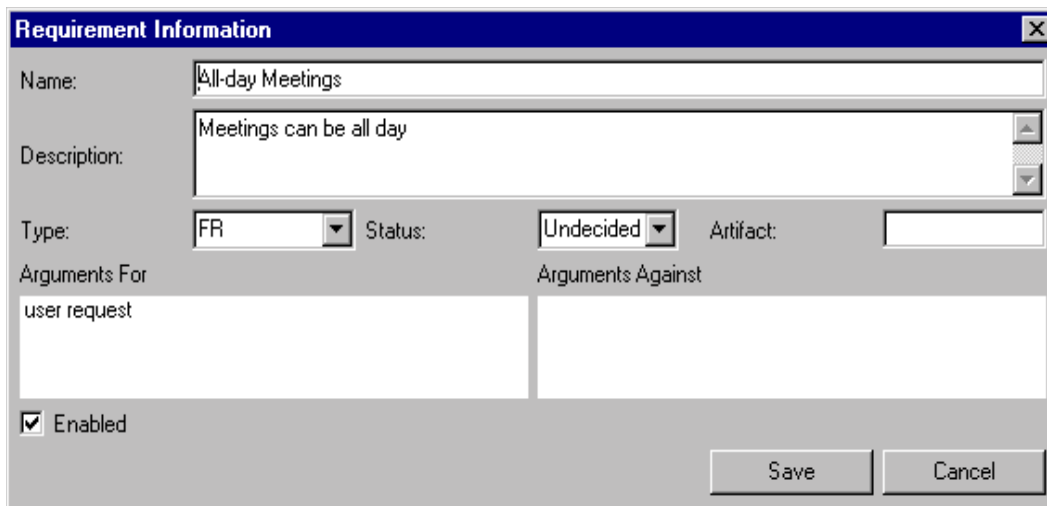
8.6. Rationale Display and Editing

Rationale can be viewed by either double-clicking on the item of interest or by selecting “Edit” from the item’s context-sensitive menu (obtained by right-clicking the item). The following sections show the edit displays for each of the rationale items.

8.6.1. Requirement

Figure 8-27 shows the Requirement Editor. The Name is mandatory and must be filled in. Other fields will have default values. The “Arguments For” and “Arguments Against” fields are for display only and will have values if this is an existing Requirement and has arguments associated with it (as is shown here). Note the box at the lower left marked “Enabled:” this is used to disable a requirement to determine the impact on the rationale. Requirements that are not yet implemented in the current release of the software but which

are planned for the future can be disabled to avoid errors being displayed in the Rationale. A disabled requirement will have a “D” superimposed on its icon.



The image shows a 'Requirement Information' dialog box. It has a title bar with a close button. The fields are: Name (text box with 'All-day Meetings'), Description (text box with 'Meetings can be all day'), Type (dropdown menu with 'FR'), Status (dropdown menu with 'Undecided'), and Artifact (text box). Below these are two text boxes for 'Arguments For' (containing 'user request') and 'Arguments Against'. At the bottom left is a checked checkbox labeled 'Enabled'. At the bottom right are 'Save' and 'Cancel' buttons.

FIGURE 8-27. Requirement Editor

8.6.2. Decision

Figure 8-28 shows the Decision Editor. As with all SEURAT elements, the Name is required. There are two types of decisions: one where sub-decisions are required and one where alternatives are required. Decisions requiring sub-decisions are ones that can be broken into sub-components where answering all the sub-decision answers the parent. In this case, alternatives are not present. The example given here shows a decision that has alternatives. The numerical evaluation for the alternative is given along with its name.

8.6.3. Alternative

Figure 8-29 shows the Alternative Editor. This gives the information about the alternative and lists the arguments for it, against it, and that have other relationships to it. These relationships refer to a dependency on another alternative being selected. The Artifact field will describe what part of the code implements this alternative.

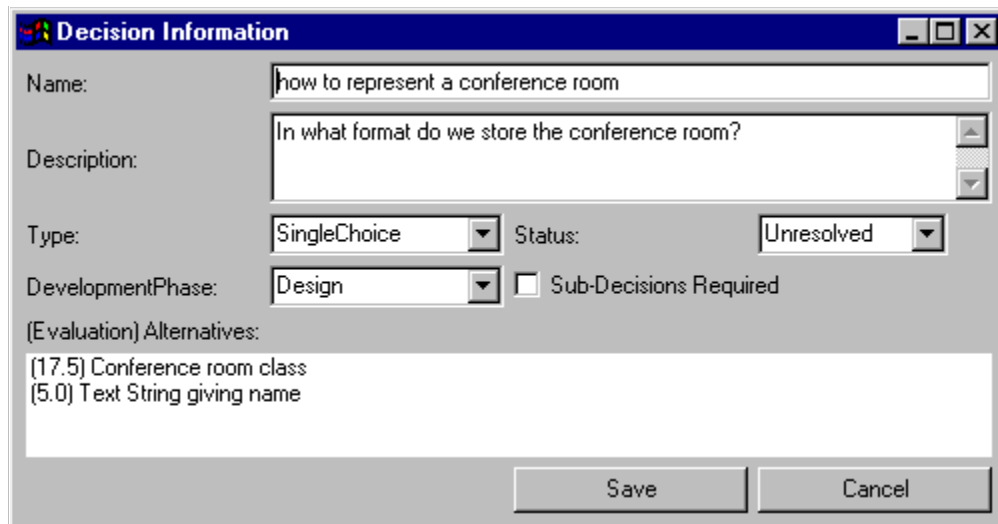


FIGURE 8-28. Decision Editor

8.6.4. Argument

Figure 8-30 shows the Argument Editor. Arguments can be associated with claims (which then point into the Argument Ontology), assumptions, requirements, or other arguments. In this example, it argues a Claim, which is shown by the Argument Type field. When an argument is initially created, it is mandatory that it be associated with something (the user will not be able to save the argument without specifying the associated element). This is done using the “Select” button. When this happens, the user is allowed to either select an already existing item to use or create a new one.

Each argument gives the type, indicating if it is for or against the alternative. The possible values vary depending on the type of the argument. These are as follows:

- *Claim* – supports or denies;
- *Requirement* – satisfies, addresses, or violates;
- *Assumption* – supports or denies;
- *Argument* – presupposes or opposes.

In addition, the user can give the Importance of the argument, the Amount (how much the alternative meets the claim), and the Plausibility (how sure they are of the argument). The Importance can be specified as default, in which case it will be inherited from the claim or Argument Ontology. Arguments involving requirements or dependencies will default to an importance of “Essential.”

The 'Alternative Information' dialog box has a title bar with a close button. It contains the following fields and controls:

- Name:** A text box containing 'user name stored as part of meeting'.
- Description:** A text box containing 'Store the name of the user as part of the meeting class.' with scroll bars.
- Status:** A dropdown menu showing 'Adopted'.
- Artifact:** A text box containing 'mOwner'.
- Arguments For:** A text box containing 'easy to code and show'.
- Arguments Against:** A text box containing 'does not provide enough information'.
- Relationships:** A large empty text box.
- Buttons:** 'Save' and 'Cancel' buttons at the bottom right.

FIGURE 8-29. Alternative Editor

The 'Argument Information' dialog box has a title bar with a close button. It contains the following fields and controls:

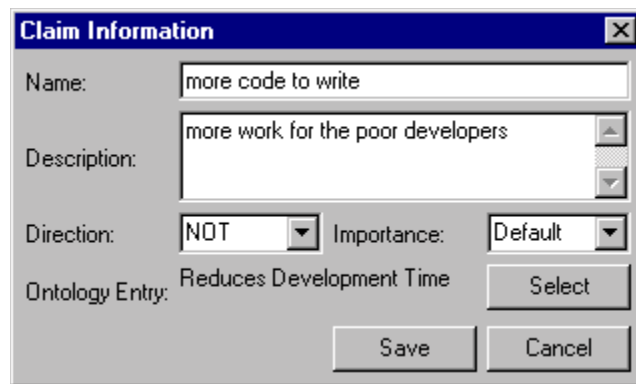
- Name:** A text box containing 'can add new attributes/methods'.
- Description:** A text box containing 'This lets you be able to extend the class to do more scheduler-specific things.' with scroll bars.
- Type:** A dropdown menu showing 'Supports'.
- Plausibility:** A dropdown menu showing 'High'.
- Amount:** A dropdown menu showing '10'.
- Importance:** A dropdown menu showing 'Default'.
- Argues:** A text box containing 'can extend our own class easily' with scroll bars.
- Argument Type:** A dropdown menu showing 'Claim'.
- Buttons:** 'Select', 'Save', and 'Cancel' buttons at the bottom right.

FIGURE 8-30. Argument Editor

8.6.5. Claim

Figure 8-31 shows the Claim Editor. This is similar to the Argument Editor but with fewer fields. The Direction indicates if the claim is that the alternative does what the ontology entry says, such as “IS” Reduces Development Time, or that the alternative does not do what the ontology entry says, as shown here by “NOT” Reduces Development Time. The user can also specify an importance here or inherit it from the Argument Ontology.

When a claim is created the user must associate an ontology entry with it. This is done using the “Select” button. This will bring up the ontology so the user can choose an entry to associate.



The screenshot shows a dialog box titled "Claim Information" with a close button (X) in the top right corner. The dialog contains the following fields and controls:

- Name:** A text input field containing the text "more code to write".
- Description:** A text input field containing the text "more work for the poor developers".
- Direction:** A dropdown menu currently set to "NOT".
- Importance:** A dropdown menu currently set to "Default".
- Ontology Entry:** A text field displaying "Reduces Development Time".
- Buttons:** A "Select" button is positioned to the right of the "Ontology Entry" field. At the bottom of the dialog are "Save" and "Cancel" buttons.

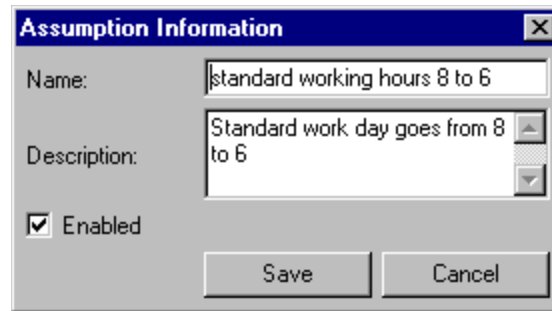
FIGURE 8-31. Claim Editor

8.6.6. Assumption

Figure 8-31 shows the Assumption Editor. This only requires a Name although it is more descriptive if a Description is specified as well.

8.6.7. Question

Figure 8-33 shows the Question Editor. For each question, there is the Status that indicates if it is answered or not, a Procedure that describes the steps that must be taken to get the answer, and the Answer (once known).

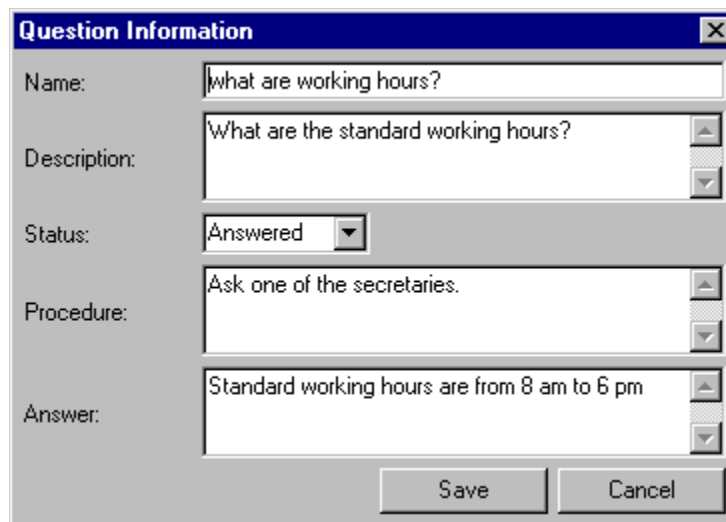


The 'Assumption Information' dialog box has a title bar with a close button. It contains a 'Name' field with the text 'standard working hours 8 to 6', a 'Description' text area with 'Standard work day goes from 8 to 6', an 'Enabled' checkbox which is checked, and 'Save' and 'Cancel' buttons at the bottom.

FIGURE 8-32. Assumption Editor

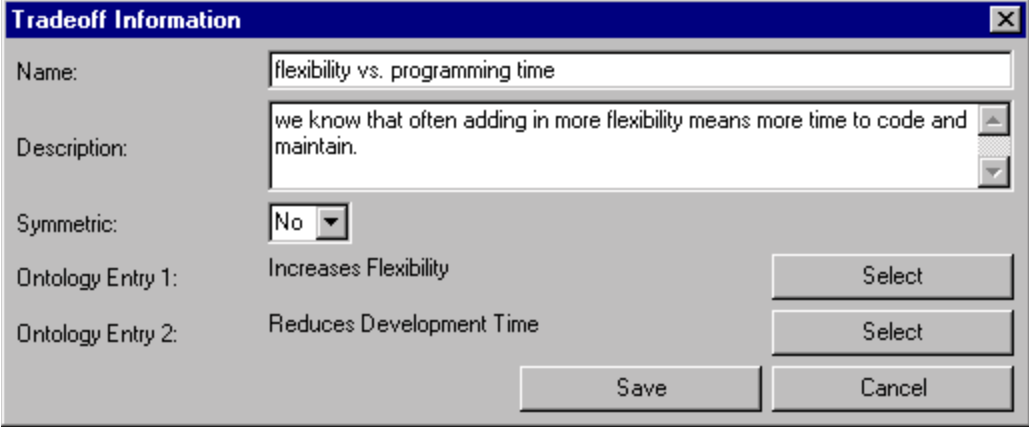
8.6.8. Tradeoff

Figure 8-34 shows the Tradeoff Editor. Tradeoffs are made between two Ontology Entry items. Tradeoffs can be symmetric, which indicates that they are always traded off against each other, or non-symmetric, which means the dependency is one-way. For example, in this non-symmetric tradeoff, Ontology Entry 1, Increases Flexibility, always needs to be traded off against Ontology Entry 2, Reduces Development Time. This means that if a choice increases flexibility it will increase development time. The other way around, however, is not true – if a choice increases development time it is not necessarily because of added flexibility.



The 'Question Information' dialog box has a title bar with a close button. It contains a 'Name' field with 'what are working hours?', a 'Description' text area with 'What are the standard working hours?', a 'Status' dropdown menu set to 'Answered', a 'Procedure' text area with 'Ask one of the secretaries.', and an 'Answer' text area with 'Standard working hours are from 8 am to 6 pm'. 'Save' and 'Cancel' buttons are at the bottom.

FIGURE 8-33. Question Editor



Tradeoff Information

Name: flexibility vs. programming time

Description: we know that often adding in more flexibility means more time to code and maintain.

Symmetric: No

Ontology Entry 1: Increases Flexibility Select

Ontology Entry 2: Reduces Development Time Select

Save Cancel

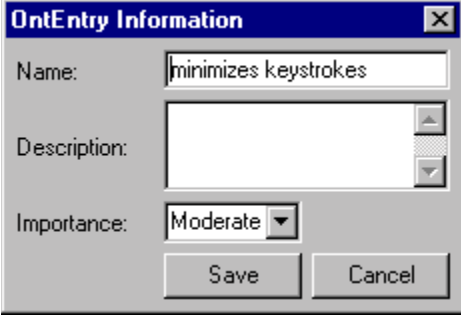
FIGURE 8-34. Tradeoff Editor

8.6.9. Co-occurrence

The Co-occurrence Editor is identical in content to the Tradeoff Editor.

8.6.10. Ontology Entry

Figure 8-35 shows the Ontology Entry Editor. This describes the entry and gives its Importance. This Importance will be inherited by any claims that reference the ontology entry.



OntEntry Information

Name: minimizes keystrokes

Description:

Importance: Moderate

Save Cancel

FIGURE 8-35. Ontology Entry Editor

8.7. Rationale Query Interface

At the top of the Rationale Explorer there is a downward arrow that allows the user to bring down a menu of query options (the Rationale Query Menu). These items are:

- *Find Rationale Entity* – used to search for a particular type of rationale entity.
- *Find Common Arguments* – used to display a list of the arguments (Claims and Ontology Entries) used in the rationale.
- *Find Requirements* – used to display a list of requirements with a particular status (such as all violated requirements). These are only shown for requirements that are used as arguments for or against selected alternatives.
- *Find Status Overrides* – used to display a list of status messages overridden by the user (i.e., removed from the Rationale Task List).
- *Find Importance Overrides* – used to display a list of Claims and Arguments that have an importance other than “Default.”

The following sub-sections describe each of these options.

8.7.1. Find Rationale Entity

The Find Rationale Entity option allows the user to search for particular types of rationale entities (requirements, decisions, alternatives, etc.). The user is first instructed to specify the type of entity as shown in Figure 8-36.

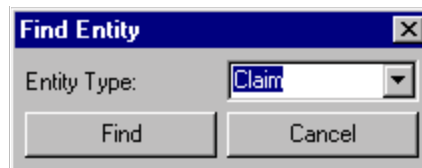


FIGURE 8-36. Find Entity Display

This then brings up a list of items of that type, as shown in Figure 8-37. The user can search for all or part of the item name to find it in the list. After find it, the user then can bring up the item in an editor by using the “Edit” button or expand the hierarchy in the Rationale Explorer to show the item in context.

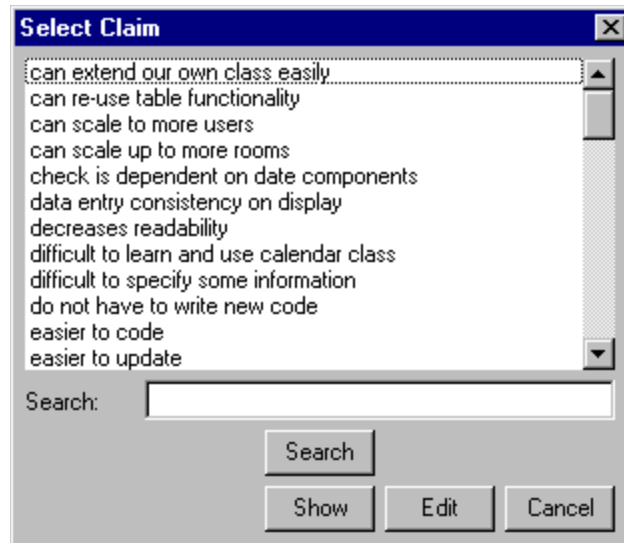


FIGURE 8-37. Select Claim Display

8.7.2. Find Common Arguments

Another useful query is to find out what the most common arguments are. This can be done for each of the three types: argument, claim, and ontology entry. Selecting which type is the first step, as shown in Figure 8-38. The user can also indicate if they are only interested in common arguments for selected alternatives.

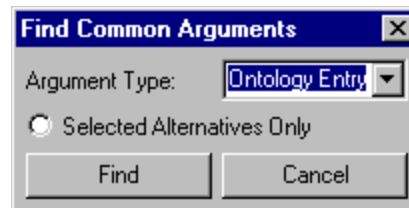
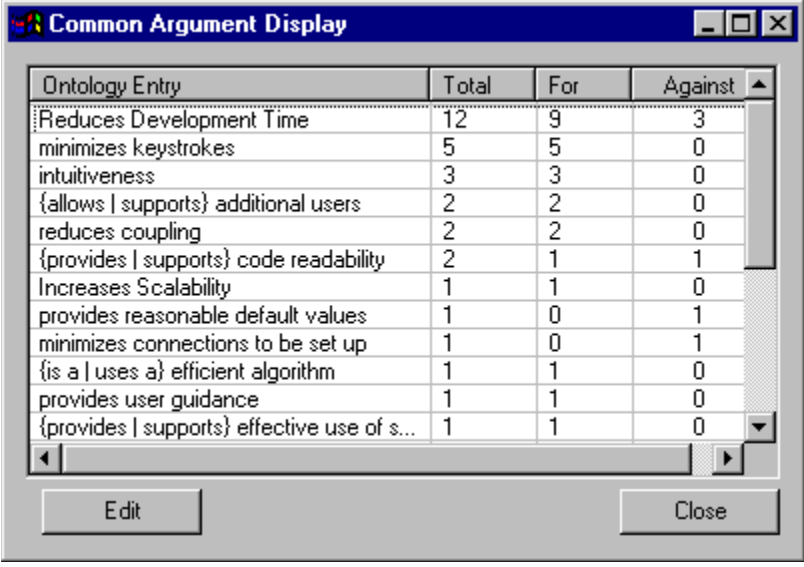


FIGURE 8-38. Find Common Arguments

After selecting the type, the arguments are then displayed in a table giving the total references, the number of times it was used to argue in support for an alternative and the number of times it opposed an alternative. Figure 8-39 shows the Common Argument Display showing ontology entries.




The dialog box titled "Common Argument Display" contains a table with four columns: "Ontology Entry", "Total", "For", and "Against". It lists various arguments with their respective counts. Below the table are "Edit" and "Close" buttons.

Ontology Entry	Total	For	Against
Reduces Development Time	12	9	3
minimizes keystrokes	5	5	0
intuitiveness	3	3	0
{allows supports} additional users	2	2	0
reduces coupling	2	2	0
{provides supports} code readability	2	1	1
Increases Scalability	1	1	0
provides reasonable default values	1	0	1
minimizes connections to be set up	1	0	1
{is a uses a} efficient algorithm	1	1	0
provides user guidance	1	1	0
{provides supports} effective use of s...	1	1	0

FIGURE 8-39. Common Argument Display

8.7.3. Find Requirements

The Find Requirements query lets the user look for requirements by their status. For example, they could get a list of all the violated, satisfied, or addressed requirements. The user selects the type using the display shown in Figure 8-40.



The dialog box titled "Find Requirements" has a "Requirement Status:" label followed by a dropdown menu currently set to "Addressed". Below the dropdown are "Find" and "Cancel" buttons.

FIGURE 8-40. Find Requirements Display

The list of requirements is shown in Figure 8-41. The user can either edit the requirement or expand it in the Rationale Explorer hierarchy to see it in context.

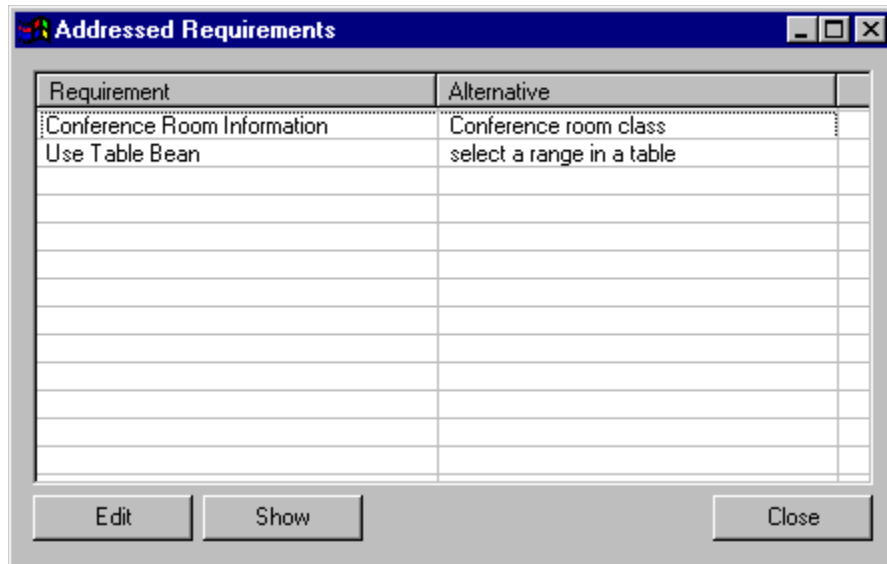


FIGURE 8-41. Addressed Requirements

8.7.4. Find Status Overrides

The user can choose to override any of the items given in the Rationale Task List. This will keep the error (or warning) from being displayed in the list or indicated by an error (or warning) icon in the Rationale Explorer. The list of overridden items can be shown by choosing “Find Status Overrides” in the Rationale Query menu. Figure 8-42 shows the Status Overrides display. The user can remove any override from this list and the Rationale Task List and Rationale Explorer will be updated when they exit from the display.

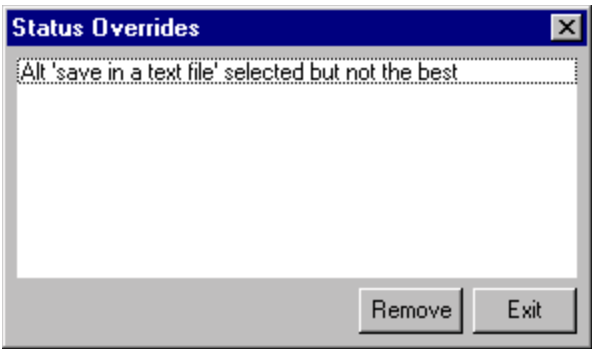


FIGURE 8-42. Status Override Display

8.7.5. Find Importance Overrides

The user can also display a list of all claims and arguments where the default importance has been overridden. Importance is overridden when a specific importance is chosen rather than leaving the importance set as “Default.” This can happen in several ways:

- A claim (which could be used by many arguments) has been given an importance other than the default inherited from the Argument Ontology.
- An argument that refers to a claim has been given an importance other than the default inherited from the claim (which inherits from the Argument Ontology)
- An argument that refers to a requirement has been given an importance of something other than “Essential.”

This display is brought up by selecting “Find Importance Overrides” from the Rationale Query Menu. Figure 8-43 shows the Importance Override Display.

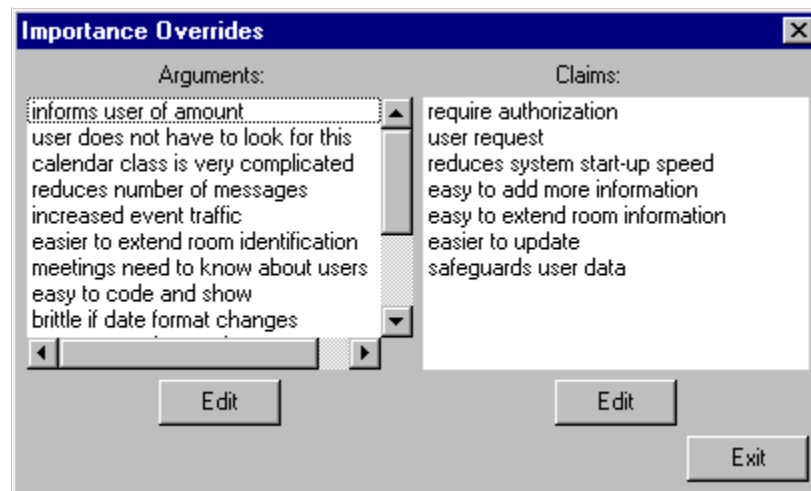


FIGURE 8-43. Importance Override Display

The following sections present a demonstration of the SEURAT system. All examples use the Conference Room Scheduling system described earlier in this dissertation. This chapter gives software maintenance examples (Section 9.1) and some inferencing examples (Section 9.2).

9.1. Software Maintenance Examples

The following sub-sections give examples of three types of software maintenance tasks: adaptive maintenance, where a change is made that does not add functionality to the system; corrective maintenance, where SEURAT assists in fixing a defect in the code; and enhanceive maintenance, where SEURAT assists in adding a new feature to the system.

9.1.1. Adaptive Maintenance

The adaptive maintenance task concerns the persistent storage of user information. The rationale contains a decision “how to store user information” where the two choices are “save in a text file” and “serialize user information.” Figure 9-1 shows the Rationale Explorer with the decision highlighted and expanded.

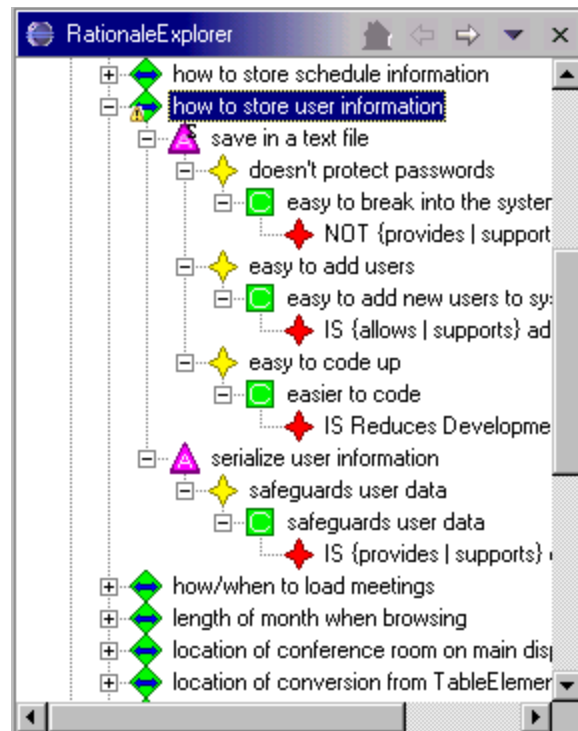


FIGURE 9-1. Rationale for “how to store user information”

The decision is shown with a yellow warning icon on it because the selected alternative, “save in a text file”, was not the best choice. This is also shown in the Rationale Task List in Figure 9-2.

The Rationale Task List window contains a table with a single column titled 'Description'. It lists five tasks, each preceded by a yellow warning icon. The last task, 'Alt 'save in a text file' selected but not the best', is highlighted with a blue background.

Description
Alt 'just the meeting name' selected but not the best
Alt 'check overlap first' selected but not the best
alt 'create new date class (meetingDate)' violates tradeoff 'Increases Flexibili
Alt 'display meetings for current week' selected but not the best
Alt 'save in a text file' selected but not the best

FIGURE 9-2. Warning Message for “save in a text file”

If the decision is edited, the arguments for and against it, and their evaluation, are shown to the user. Figure 9-3 shows the decision and that the selected alternative “save in a text file” has a negative value, while the alternative “serialize user information” has a positive one.

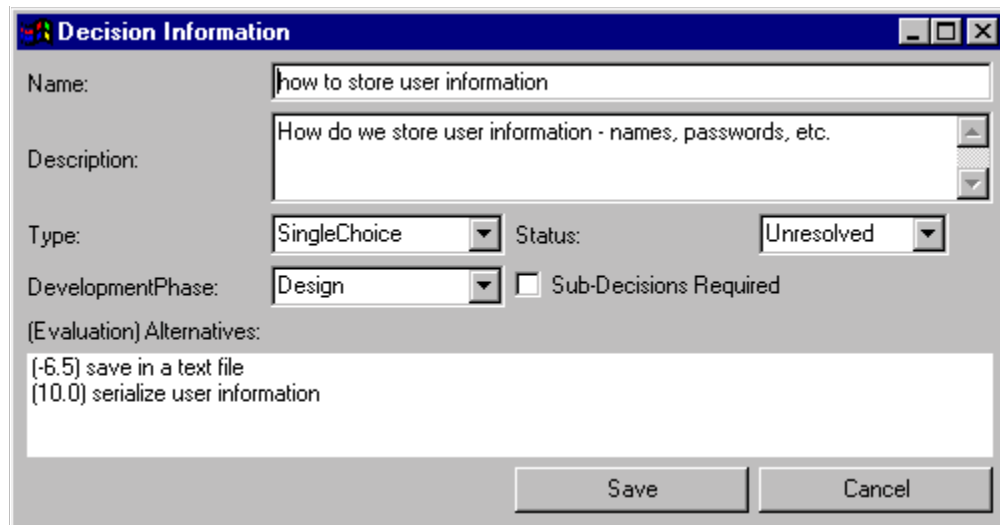


FIGURE 9-3. Decision “how to store user information”

Displaying the alternative “serialize user information” gives additional information about that choice and tells how it can be implemented. Figure 9-4 shows the alternative and the description that tells the maintainer that they can use `ObjectOutputStream` to serialize the user information.

The next step is to find the place in the code where the information is stored. A logical place to start is to find where the currently selected alternative, “save in a text file”, is implemented. Editing the alternative (see Figure 9-5) shows that it is associated with the method “actionPerformed.” If this name is not familiar to the maintainer, they can find the code via the Bookmarks Display. Figure 9-6 shows the Bookmarks Display showing the bookmark that maps the alternative to the code. Double-clicking on that alternative brings up the appropriate code in the editor, shown in Figure 9-7. The code that does the saving is further down in the method but can easily be found by the maintainer.

The maintainer can also use SEURAT to find the code where the user information is loaded. Figure 9-8 shows the Rationale Explorer with a decision on “where to load user information.” The selected alternative is “load from Login Users class”

and the maintainer can either find this class in the Eclipse Package Explorer or use the bookmarks as described above.

After making the necessary changes, the maintainer should go back to the rationale and select the new alternative, associate it with the appropriate code, and de-select the old one. This is not currently enforced. When changing the status for the two alternatives, the maintainer has the opportunity to enter a reason for the status change. The maintainer should also remove the association to the code from the alternative that is no longer selected.

The screenshot shows a dialog box titled "Alternative Information". It contains the following fields and controls:

- Name:** A text box containing "serialize user information".
- Description:** A text box containing "Use an ObjectOutputStream (and InputStream) to serialize the user information".
- Status:** A dropdown menu currently showing "At_Issue".
- Artifact:** An empty text box.
- Arguments For:** A text box containing "safeguards user data".
- Arguments Against:** An empty text box.
- Relationships:** An empty text box.
- Buttons:** "Save" and "Cancel" buttons at the bottom right.

FIGURE 9-4. Alternative “serialize user information”

Alternative Information

Name: save in a text file

Description: Save user names and passwords in an ASCII text file

Status: Adopted

Artifact: actionPerformed

Arguments For

- easy to code up
- easy to add users

Arguments Against

- doesn't protect passwords

Relationships

Save Cancel

FIGURE 9-5. Alternative “save in a text file”

Description	Resource	In Folder
Alt: 'only owner can cancel meet...	Meeting...	CRSEnhancive/meeting
Alt: 'override equals'	Meeting...	CRSEnhancive/meeting
Alt: 'room and building combined'	Conferen...	CRSEnhancive/sched...
Alt: 'save in a text file'	CRSEnh...	CRSEnhancive/sched...
Alt: 'select a range in a table'	CRSEnh...	CRSEnhancive/sched...
Alt: 'send individually'	Meeting...	CRSEnhancive/meeting

Error Log Console Bookmarks Rationale Task List Tasks Search

FIGURE 9-6. Bookmark Showing Association for “save in a text file”

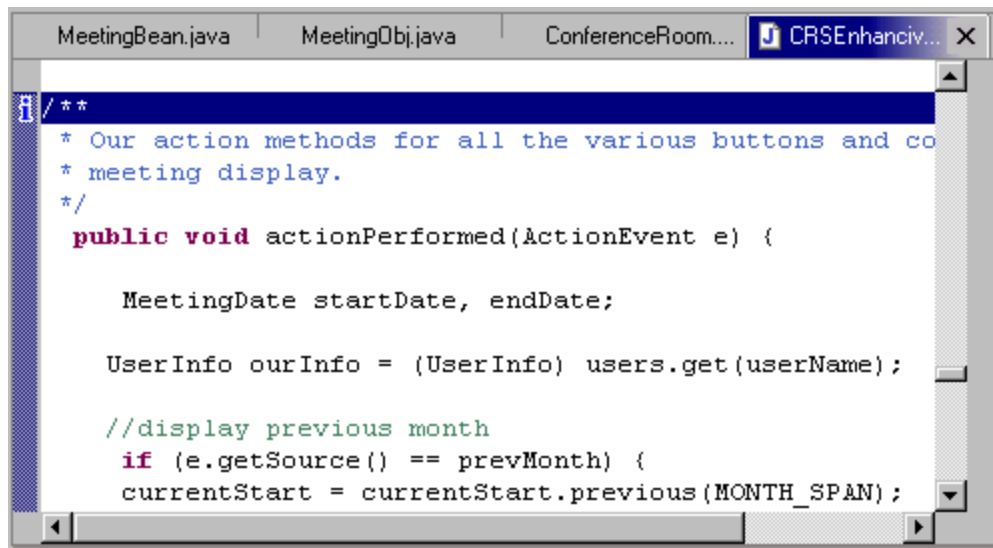


FIGURE 9-7. Method Where User Information is Saved

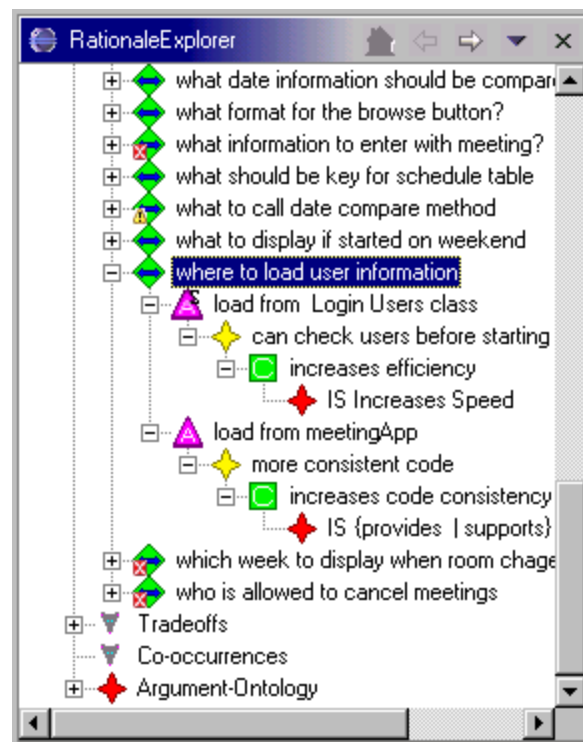


FIGURE 9-8. Rationale for “where to load user information”

9.1.2. Corrective Maintenance

The corrective maintenance task consists of fixing a bug in the Conference Room Scheduling system that occurs when the user changes which conference room is displayed. The system should show a schedule for the new room for the same week that they were looking at before changing rooms but instead it goes back to show the schedule for the current week.

The rationale, shown in Figure 9-9, indicates that the decision about what room to display when the room changes has an error. This is indicated by the red icon overlaying the decision icon. The Rationale Task list, shown in Figure 9-10, explains that the alternative that was selected, “display meetings for current week”, has arguments against it but none for it.

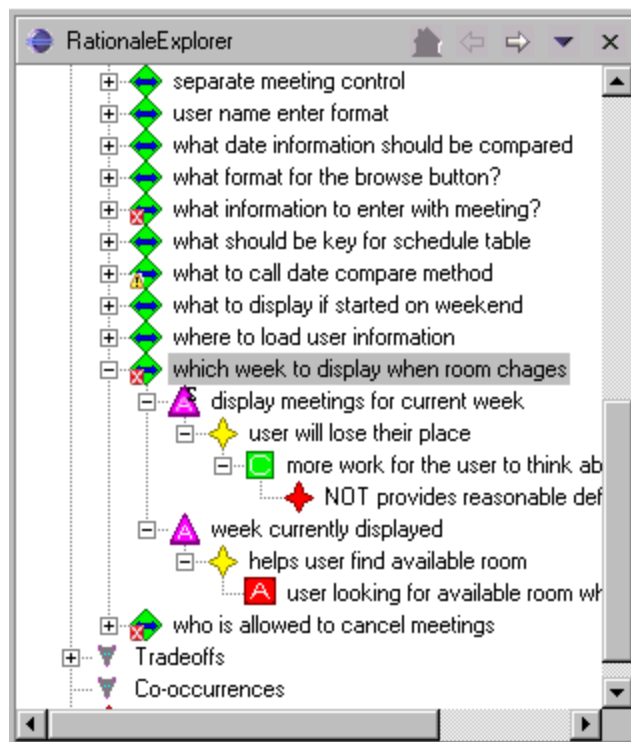


FIGURE 9-9. Rationale for “which week to display when room changes”

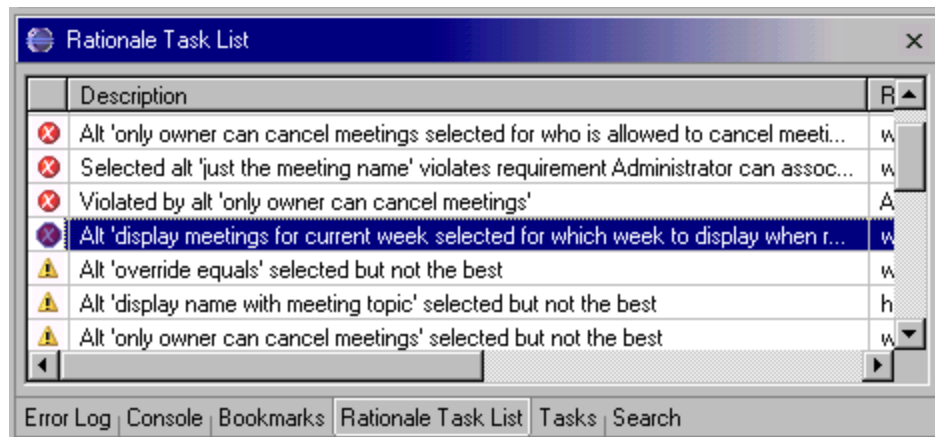


FIGURE 9-10. Error for the Selected Alternative

The user can find the code that selects which week is displayed by either editing the alternative to get the name of the code element or, more quickly, by finding the selected alternative in the Bookmark Display, shown in Figure 9-11. Double-clicking on the bookmark brings up the code in the editor, shown in Figure 9-12. After making the required change, the maintainer should change the rationale to select the correct alternative and make sure it is associated with the code correctly.

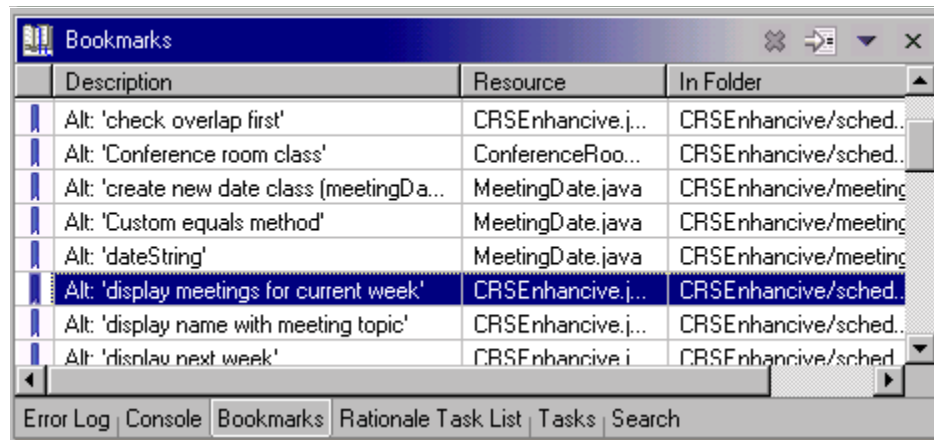


FIGURE 9-11. Bookmark Display Showing Association for “display meetings for current week”

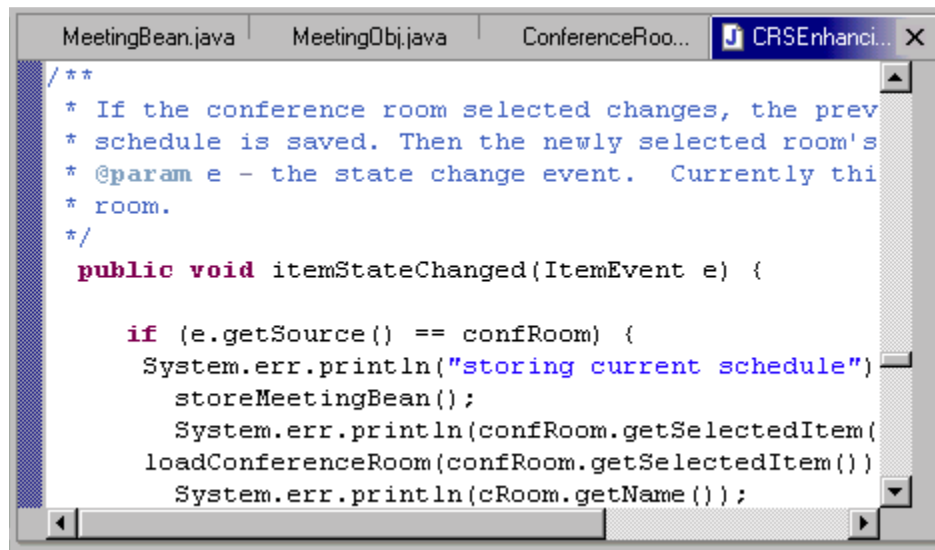


FIGURE 9-12. Method Where Dates are Reset

9.1.3. Enhance Maintenance

The enhance maintenance task involves adding a new feature to the system. In this case, it involves implementing a requirement that had initially been deferred: allowing users with administrative privileges to cancel meetings scheduled by other Conference Room System users. This is necessary in case a meeting has to be moved due to a higher priority meeting and the originating user is not around to do it.

When the requirement “Administrator can cancel any meeting” is enabled, it shows up as having been violated in two places in SEURAT. The requirement itself, as shown in the Rationale Explorer in Figure 9-13, has an error icon over it. There is also an error message on the Rationale Task Display shown in Figure 9-14.

When the maintainer looks at the decision list in SEURAT, there are several that have error icons on them. One is the decision “who is allowed to cancel meetings” shown in Figure 9-15. This has three alternatives, only one of which does not violate any requirements. That alternative is “owner or administrator can cancel meetings.”

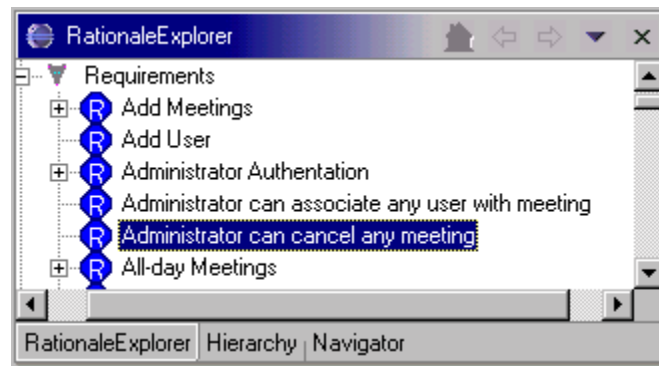


FIGURE 9-13. Violated Requirement “Administrator can cancel any meeting”

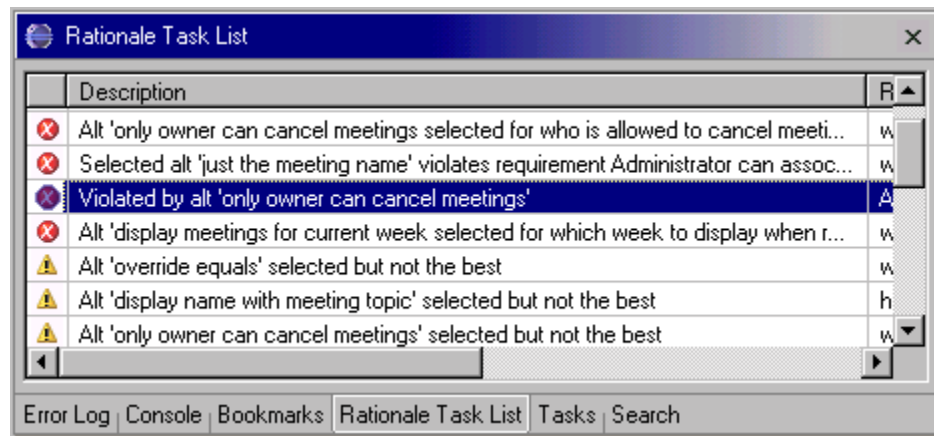


FIGURE 9-14. Error Showing Violated Requirement

The maintainer then needs to find the place in the code that checks if the meeting can be cancelled or not. This can be done the most quickly using the Bookmarks Display. Figure 9-16 shows the Bookmarks Display with the alternative “only owner can cancel meetings” highlighted. When double-clicked, the editor comes up with the appropriate code, as shown in Figure 9-17. When done with the changes, the maintainer needs to update the rationale so that the correct alternative is selected and the code associations are also correct.

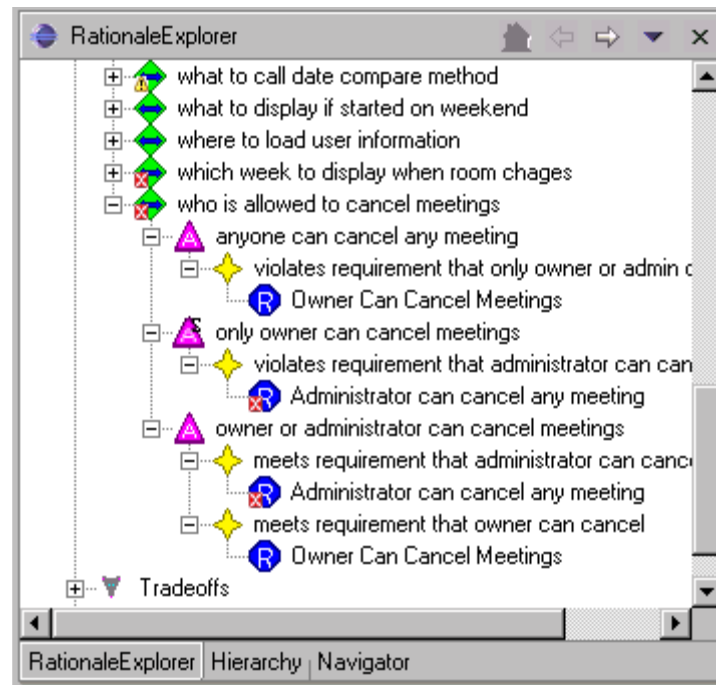


FIGURE 9-15. Rationale for “who is allowed to cancel meetings”

Description	Resource	In Folder
Alt: 'Main application class'	CRSEnhance.j...	CRSEnhance/sched..
Alt: 'meeting bean controls all'	MeetingBean.java	CRSEnhance/meeting
Alt: 'only owner can cancel meetings'	MeetingBean.java	CRSEnhance/meeting
Alt: 'override equals'	MeetingDate.java	CRSEnhance/meeting
Alt: 'room and building combined'	ConferenceRoo...	CRSEnhance/sched..
Alt: 'save in a text file'	CRSEnhance.j...	CRSEnhance/sched..
Alt: 'select a range in a table'	CRSEnhance.j...	CRSEnhance/sched..
Alt: 'send individually'	MeetingBean.java	CRSEnhance/meeting

The bottom of the window shows tabs for 'Error Log', 'Console', 'Bookmarks', 'Rationale Task List', 'Tasks', and 'Search'.

FIGURE 9-16. Bookmark Showing Association for “only owner can cancel meetings”

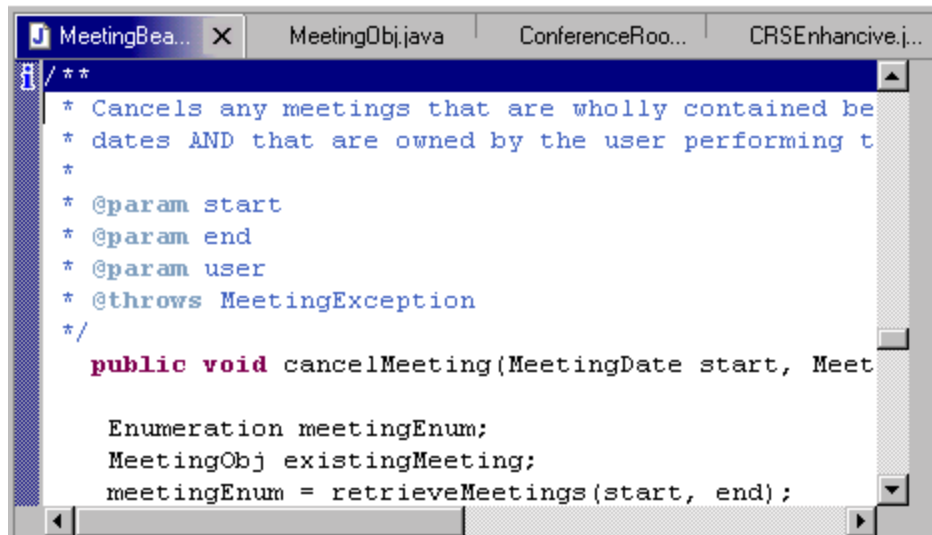


FIGURE 9-17. Method Where Meetings are Cancelled

9.2. Inferencing Examples

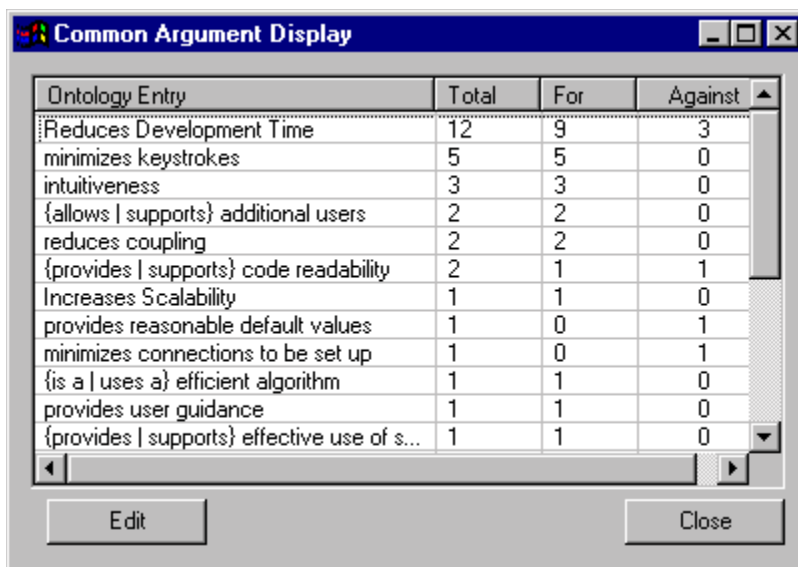
SEURAT also lets the maintainer do “what-if” inferencing to see the effect of certain changes on the system being maintained. The following sections demonstrate two ways this can be done: changing design priorities and disabling an assumption. The maintainer can also perform “what-if” by disabling requirements but that is not shown here.

9.2.1. Changing Priorities

Design priorities are expressed in the rationale by the importance given to the items in the Argument Ontology. SEURAT allows arguments and claims to inherit importance values from the Argument Ontology. These importance values are used to evaluate each alternative. If the importance in the ontology is modified, that value will then propagate through the rest of the rationale. This may mean that some alternatives that were selected may no longer be the best choice.

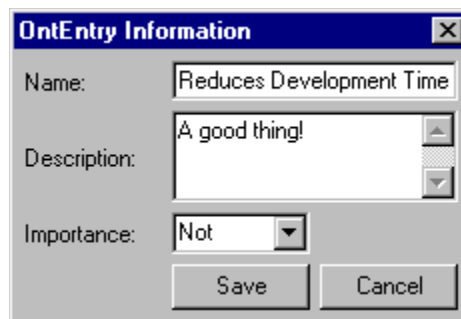
For this example, we first looked to see what items in the Argument Ontology were appearing the most in arguments concerning selected alternatives. Figure 9-18 shows the Common Argument Display with Ontology Entries listed in the order of the number of references. The one that appears the most often is “Reduces Development Time.” We then changed the importance value of “Reduces Development Time” from “Moderate” to “Not” as shown in Figure 9-19. This

caused all the claims and arguments that inherited their importance values from it to be re-evaluated. This resulted in several decisions where the best alternative was no longer selected. The new warning messages generated are shown in the Rationale Tasks Display given in Figure 9-20. Note, for purposes of this example all other errors and warnings were overridden so they would not appear on the display. This capability is very useful because priorities may change over the life of a system. For example, reducing development time may have been a primary concern when the system was first built because of schedule limitations, but some of those decisions may be worth re-examining when time is no longer as constrained.



Ontology Entry	Total	For	Against
Reduces Development Time	12	9	3
minimizes keystrokes	5	5	0
intuitiveness	3	3	0
{allows supports} additional users	2	2	0
reduces coupling	2	2	0
{provides supports} code readability	2	1	1
Increases Scalability	1	1	0
provides reasonable default values	1	0	1
minimizes connections to be set up	1	0	1
{is a uses a} efficient algorithm	1	1	0
provides user guidance	1	1	0
{provides supports} effective use of s...	1	1	0

FIGURE 9-18. Common Argument Display with Ontology Entries



OntEntry Information

Name: Reduces Development Time

Description: A good thing!

Importance: Not

Save Cancel

FIGURE 9-19. Ontology Entry “Reduces Development Time”

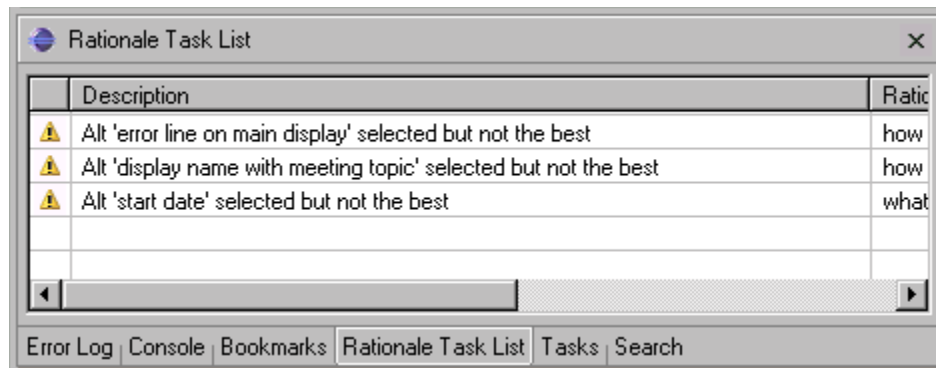


FIGURE 9-20. Rationale Task Display with New Warnings

9.2.2. Disabling Assumptions

One of the features unique to SEURAT and the RATSpeak representation is the explicit recording of assumptions. Assumptions can become invalid over time, which is a key reason for why software needs to continually evolve [Lehman, 2003]. It is important to re-examine assumptions during maintenance to ensure that they still hold. SEURAT lets the user disable an assumption to see the impact that has on the rest of the rationale.

Figure 9-21 shows a list of all assumptions captured for the Conference Room Scheduling System with the assumption “standard working hours 8 to 6” highlighted. This assumption can be edited from this display and then disabled, as shown in Figure 9-22.

Disabling the assumption causes the alternative that refers to it to be re-evaluated again. This results in a warning for the decision “schedule duration.” Figure 9-23 shows the Rationale Explorer with the disabled assumption highlighted. Figure 9-24 shows the Rationale Task List with the new warning message highlighted.

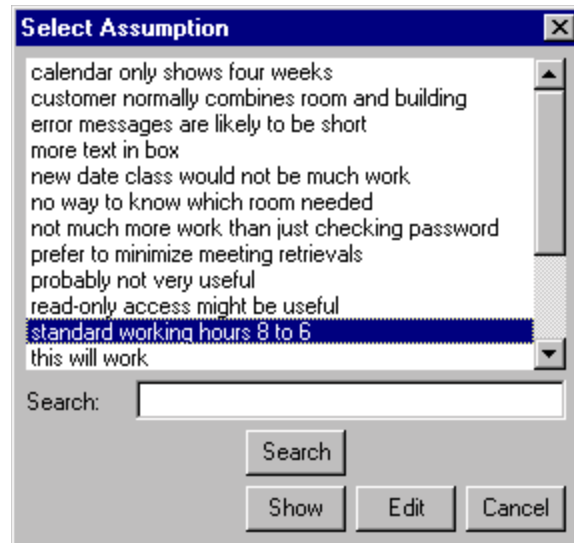


FIGURE 9-21. Assumptions for Conference Room Scheduling System

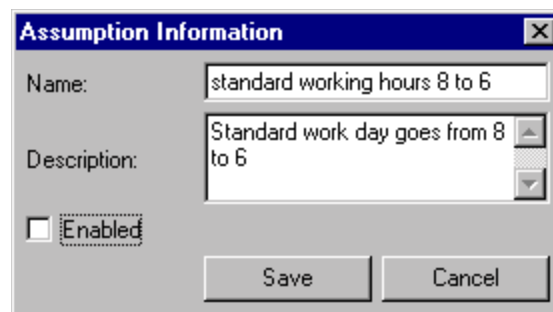


FIGURE 9-22. Assumption “standard working hours 8 to 6”

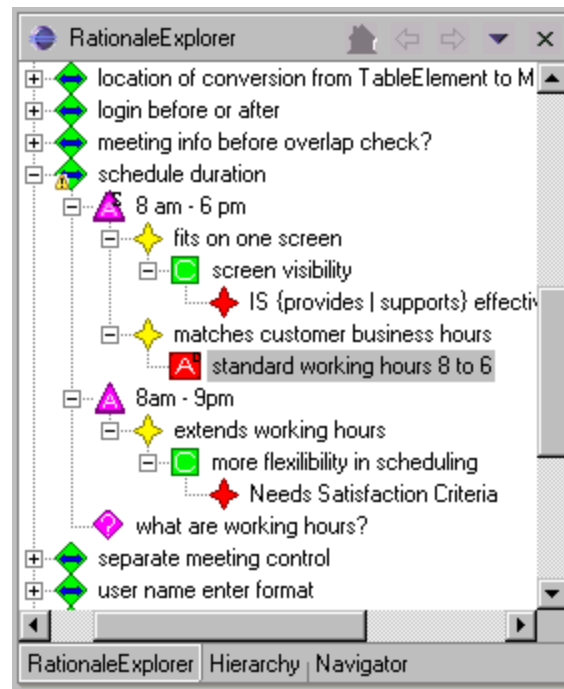


FIGURE 9-23. Rationale for Decision “schedule duration”

Rationale Task List		R
Description		
Alt 'just the meeting name' selected but not the best	w	
Alt 'check overlap first' selected but not the best	rr	
alt 'create new date class (meetingDate)' violates tradeoff 'Increases Flexibility vs...	h	
Alt 'display meetings for current week' selected but not the best	w	
Alt 'save in a text file' selected but not the best	h	
Alt '8 am - 6 pm' selected but not the best	s	

FIGURE 9-24. Rationale Task List with New Warning

9.3. Summary

This chapter gave some examples of how SEURAT could be used to assist in maintenance. These included showing SEURAT assisting in adaptive, corrective, and enhance maintenance as well as showing SEURAT performing two types of what-if inferencing where the maintainer made changes in the rationale to see the affect on the decisions already made.

This chapter describes the experiments conducted to evaluate using the SEURAT system to support software maintenance. Section 10.1 describes the experiment design by giving the overall goals for the experiment (10.1.1), a description of the experiment planning process (10.1.2), the choice of tasks for the experiment (10.1.3), and the selection/grouping of experiment subjects (10.1.4). Section 10.2 presents the time it took to perform the experiment with and without SEURAT (10.2.1), the usability results (10.2.2), the usefulness assessment (10.2.3), and user comments about the experiment (10.2.4). Section 10.3 provides the conclusions from the experiment.

10.1. Experiment Design

The following sections describe the experiment design and subject selection.

10.1.1. Experiment Goals

The first task towards creating the experiment was to determine what the overall goals were. This was done by performing a Goal Questions Metric (GQM) analysis [Basili, et. al., 1994]. This consists of determining what the goals of the evaluation are, what questions need to be answered to achieve the goal, and what metrics can be used to answer the questions. Table 10-2 shows the GQM analysis for the SEURAT evaluation.

This analysis indicated that the experiment needed to collect timing data on how long it took to complete the tasks, the perceived assistance provided by SEURAT, and SEURAT's usability. To compare the results both with and without SEURAT, the experiment needed to be designed so that there could be an experimental group using SEURAT and a control group that did not use SEURAT.

10.1.2. Experiment Design

The GQM analysis showed that the questions that the experiment had to answer referred to how long it took to perform three types of maintenance tasks: adaptive, corrective, and

enhance. Adaptive maintenance consists of making modifications to the software that do not add new functionality to the system. A typical example of an adaptive maintenance task would be refactoring. Corrective maintenance refers to maintenance changes that correct defects in the software (“fix bugs”). Enhance maintenance involves extending the functionality of the system.

Experiment design consisted of four phases. First, we needed to select a system to perform maintenance on where we had rationale available. Second, we had to define the experiment itself. Third, we needed to select the tasks that represent each type of maintenance. Finally, we needed to dry-run the candidate experiment to detect any unexpected problems with the experiment protocol, the experiment tasks, or the SEURAT system.

TABLE 10-2. GQM Analysis Results

Goal: To Evaluate SEURAT to determine its usefulness	Question	Metric
Purpose: Evaluate Issue: using Rationale to improve the <i>efficiency</i> of Object (process): software maintenance Viewpoint: from the maintainer’s viewpoint	Can the maintainer fix problems faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty
	Can the maintainer perform adaptive maintenance faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty
	Can the maintainer make enhancements to the system faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty

Purpose: Evaluate Issue: using Rationale to improve the <i>effectiveness</i> of Object (process): software maintenance Viewpoint: from the project manager's viewpoint	Is the maintainer better at detecting problems in the software when there is rationale available to help?	Number of problems found Time required to find problems
	Does the maintainer produce better solutions to problems when there is rationale available to help?	Number of errors in solutions Quality (subjective) of solutions
	Does the maintainer make better choices when performing adaptive maintenance when there is rationale available to help?	Number of successful adaptive improvements (vs. unsuccessful)
	Does the maintainer make better improvements to the system when there is rationale available to help?	Number of errors in solutions Quality (subjective) of solutions
Purpose: Evaluate Issue: the <i>usability</i> of Object(process): the SEURAT system Viewpoint: from the maintainer's viewpoint	Is the SEURAT system easy to use?	Perceived ease of use.

10.1.2.1. System Selection and Rationale Collection

Two candidate software systems were evaluated for use in the experiment. The first was a multi-player solitaire game that could be played over a network (Kombat Solitaire). Rationale was gathered for this system as part of the DR representation development. It became apparent during rationale elicitation that this system would be far too complex to give tasks that were simple enough to perform in the limited time available for the experiment. The second system was a Meeting Scheduler system that had been developed as part of a class project. This system was too simple to use but was easily extensible to a slightly more complex Conference Room Scheduler system. The Conference Room Scheduler system was then chosen to be the target system for the experiment, while the Kombat Solitaire system and a partial set of rationale generated for it were used during the experiment for SEURAT training so that the subject could learn SEURAT without being exposed to the same code that they would use for the rest of the experiment.

Rationale had already been collected for portions of the Conference Room Scheduler system during the maintenance study described in Chapter 5. This was expanded for use in the experiment, including adding new rationale specifically addressing areas of the code that were going to be involved in the experiment.

10.1.2.2. Experiment Design

The experiment was broken into several parts:

- SEURAT Tutorial – The tutorial consisted of a walkthrough of the SEURAT system and its functionality. This was done using the code and partial rationale for the Kombat Solitaire system so that the Experimental Group did not get any extra time with the Conference Room Scheduling System code. A User's Guide to SEURAT was provided at this time for use during the experiment.
- Eclipse training (if necessary – Some subjects were experienced Eclipse users).
- Conference Room Scheduler system training – The subjects were introduced to the system to be maintained by giving them a copy of the requirements and demonstrating the system's functionality.
- Adaptive Maintenance Task – The subjects were told what the task was.
- Corrective Maintenance Task – The subjects were told what the task was and the error in the system was demonstrated to them.
- Enhance Maintenance Task – The subjects were told what the task was.
- Survey on Usability, Utility, and Experiment evaluation – The survey was designed to obtain an assessment of the general usability of the system, the utility of the system, and to determine whether the maintenance task explanations were clear. Copies of these surveys are given in Appendix B.

The three maintenance tasks were given in a random order for each subject to help cancel out the effects of earlier exposure to the code on the later tasks, i.e. learning and maturation [Bratthall, et. al., 2000]. The control group did not have any SEURAT training and were given a shorter survey that did not ask about SEURAT Usability and Utility.

10.1.2.3. Experiment Task Selection

The experiment tasks were developed to cover the three types of maintenance: adaptive, corrective, and enhanceive. This allowed us to get a comparison of how long each took both with and without SEURAT. The tasks were chosen so that the experiment could be performed in under three hours, preferably under two hours. This was because many of the subjects were software professionals working full time who would be performing the experiment at the end of their workday. This was a serious constraint because it limited the complexity of the tasks given. The option of cutting off the tasks after a pre-determined length of time was considered to limit the time required for the experiment but that then makes the results more difficult to analyze because it is not clear if the subject was minutes or hours away from completing the tasks.

1. **Adaptive Maintenance:** the task chosen was to change how the username and password information for the scheduler users was stored. The initial version of the system stored and read this information from a text file (users.txt), which anyone could read and edit. The subjects were asked to change how this was done so that the information would be stored in binary.

The SEURAT subjects could look through the rationale to look for two key decisions: a decision about how to store the user information, and a decision about where in the code the user information should be loaded. Finding the key decisions is the first step toward finding the alternative chosen, which can then take the subject directly to the code that requires modification. The decision about how to store the user information had two alternatives. The alternative chosen for the decision, to save in a text file, and a better alternative, serializing the information. The serialization alternative description said that the information could be serialized using `ObjectOutputStream`.

This was a fairly involved change that needed to take place in two different parts of the code. There was more than one way the change could be implemented.

2. **Corrective Maintenance:** to support this task, a bug was inserted into the software that would display the schedule for the current week when the user selected a different conference room. Displaying the current week is a bug because often a user would be looking to schedule a meeting in the future and would want to see what was available for that date. If they had moved ahead in the schedule, they shouldn't have to do that again when the room changes since they probably want to see the same week for the different room.

The fix for this defect is simple and involved removing a line of code that reset the

dates. The code was commented to point out what that line of code did. The comments were added after subjects had trouble with the task during dry runs of the experiment. The rationale included a decision about what room to display when the room changes. There were two alternatives captured in the rationale for that decision. One alternative was the chosen one of displaying the current date (the bug) and the other kept the dates the same when changing rooms. The SEURAT group could use this to find the code that needed to be changed.

3. **Enhance Maintenance:** this task involved adding functionality to the system. The scheduling system supported two types of users: regular users and those with administrative privileges. Administrative users could edit user information. The enhancement was to also allow administrative users to cancel meetings scheduled by any other user. This would be necessary if a meeting had to be moved or canceled and the user who scheduled it was not around to cancel it. The code change required two things: adding a check to see if the user had administrative privileges and passing that information into the method that performed the check. The rationale had a decision about who was allowed to cancel meetings and this rationale pointed to the code where the check needed to be made.

The subjects had unexpected difficulty with this during the dry runs (described in the next section) so the call to the cancel method was modified to make it obvious that the caller had access to the administrative information but just chose to not pass it in.

10.1.2.4. Experiment Dry Runs

Experiment dry runs were then performed with two subjects of different levels of experience: one at the expert level and one at the moderate level. This was done to gain practice administering the experiment, to get timing information, and to obtain initial insight into what some problems might be in the experiment and in SEURAT so that they can be corrected prior to the start of the actual experiments. These dry runs proved to be invaluable.

First, the SEURAT training took more time than initially anticipated. The original estimate was 10 minutes, but the actual training took between 30 and 40 depending on the number of questions from the subject. The training also needed to emphasize which parts of the rationale were associated with the code, i.e. the decisions and alternatives. It was very common during the dry run, and again during the experiment, for the subjects to fixate on the system requirements (which were something that they were familiar with) rather than the decisions and alternatives as places to look to find the solutions to the tasks.

Second, the tasks needed to be drastically shortened. This was done between the two dry runs since the expert Java programmer took far more time than was expected, indicating that moderate or inexperienced Java programmers would have difficulty completing the experiment in a reasonable amount of time. Timing became especially critical because the SEURAT tutorial took longer than expected. These adjustments were made in two ways. One was by simplifying the tasks. The other was by adding additional comments (hints) to the code describing how to complete the assignment and making other adjustments that were designed to help lead the subject to a solution.

Third, there were a couple of usability problems with the system. One was that the bookmarks for the system were too hard to read without a great deal of scrolling. Since the bookmarks were the easiest way to navigate from rationale to the code it was crucial that they be readable. There were similar readability issues with the error messages in the Rationale Task Display. All of these messages were shortened, clarified, and made more consistent with each other. Another discovery was that both subjects tried to edit rationale entities by double-clicking on them in the hierarchy display. In retrospect, this is an obvious thing to be doing since that is how you would edit files in an IDE. It was a simple change to bring up the editor in response a double-click.

The dry runs proved to be invaluable as they indicated several significant problems with both the experiment tasks and how I was administering them. Adjustments were made prior to doing the actual experiments with the subjects.

10.1.3. Experiment Subject Selection

Experiment subjects were found by e-mailing WPI faculty and graduate students and by asking personnel at Charles River Analytics (www.cra.com) to participate. Funding was provided by WPI so WPI community members were given the first opportunity to participate. The subjects were given a questionnaire, shown in Appendix B, that was used to get their contact information, level of Java expertise (Some (S), Moderate (M), or Expert (E)), work experience, and Eclipse experience. They were also asked to describe their last three Java projects as a means of cross-checking their Java experience level. One subject described themselves as a Java expert but only listed some simple class projects so was downgraded to the moderate group.

Twenty subjects expressed interest and were divided into the two groups of 10: SEURAT and Control. The groups were structured to balance Java expertise and work experience as much as possible. Table 10-3 summarizes the SEURAT and Control groups.

TABLE 10-3. Experimental Group Summary

	SEURAT	Control
Java Experience (E/M/S)	3/4/3 people, respectively	3/4/3 people, respectively
Average Work Experience	6.85 years	5.65 years
Eclipse Experience	60%	60%

10.2. Experiment Results

Experimental data was collected in several ways. The first was by timing the maintenance tasks, the second was using a usability survey, and the third was a usefulness survey. These are described in the following sections.

10.2.1. Support for Maintenance

The two types of timing data were captured for each of the three maintenance tasks. First, the time to initially find the change was recorded. This was done because this was the portion of the task where SEURAT was expected to be of the most assistance. Second, the total time to complete the task was recorded. The following sections give the results for each of the three tasks.

10.2.1.1. Adaptive Maintenance

This was the most difficult of the three tasks, especially for the subjects who had only some Java experience. It involved making changes in two parts of the code: one to write out the user information and the other to read in the user information. Figure 10-1 shows the average times to find the first of the two places to change. For some users this was the write, for others the read. Figure 10-2 shows the average time to complete the task. Using SEURAT decreased the time to find and make the changes for the Moderate and Some Java user groups but did not improve the performance for the Expert Java user.

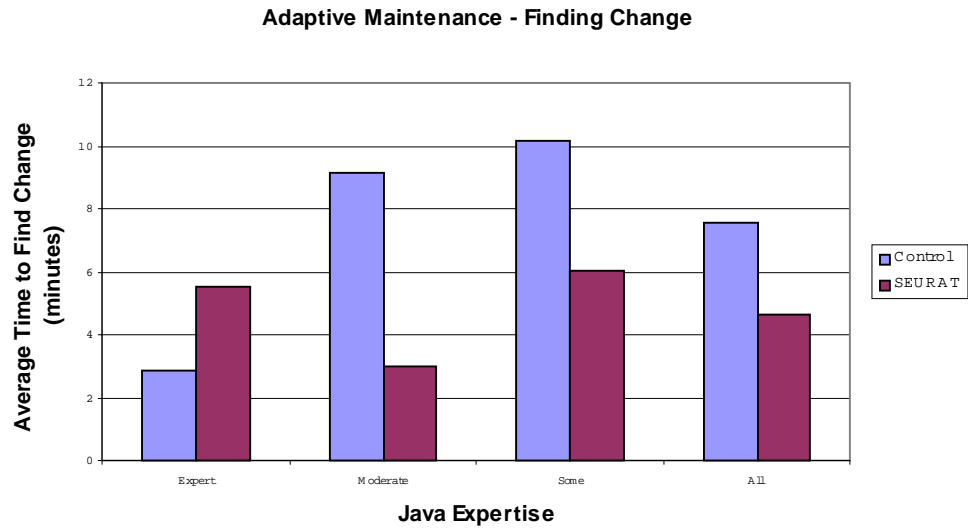


FIGURE 10-1. Adaptive Maintenance - Time to Find Change

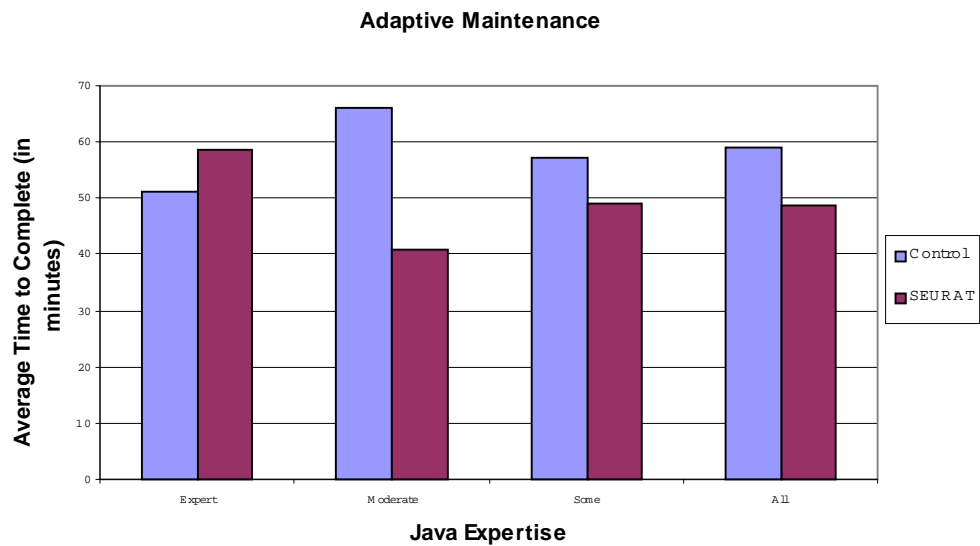


FIGURE 10-2. Adaptive Maintenance - Time to Complete Task

One interesting observation is that the variance in times is much smaller for the SEURAT group than for the control group. Figure 10-3 shows the variance for finding the first place

to change and Figure 10-4 shows the variance for the time to complete the task. Plots showing the times to find the change and complete the task are plotted versus the number of years of subject experience in Figure 10-5 and Figure 10-6.

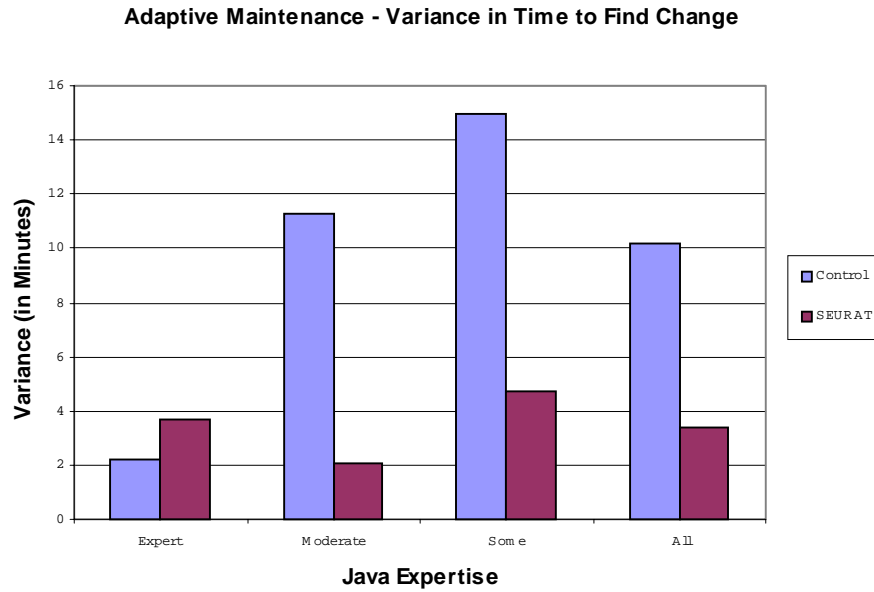


FIGURE 10-3. Adaptive Maintenance - Variance in Time to Find the Change

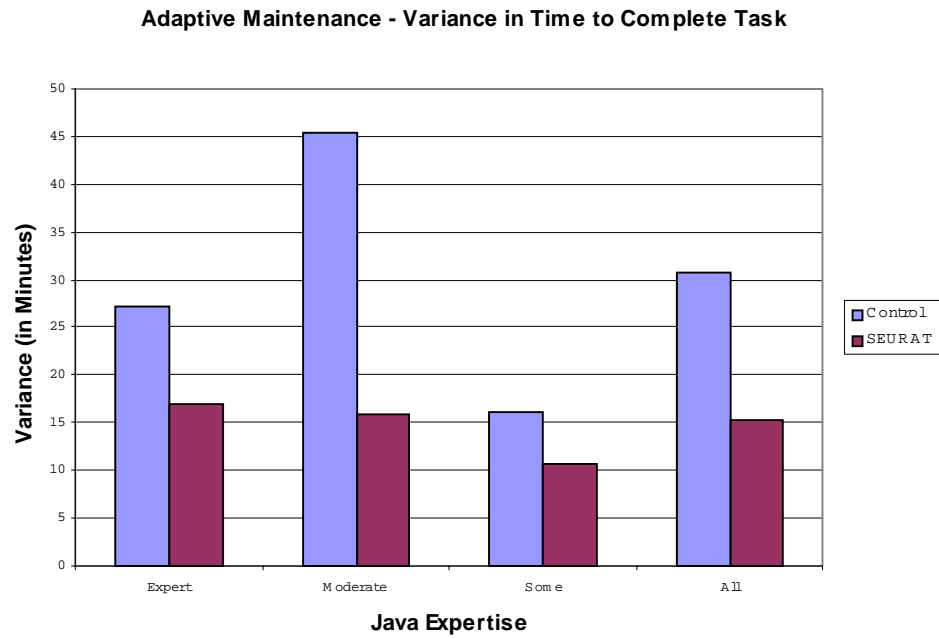


FIGURE 10-4. Adaptive Maintenance - Variance in Time to Complete Task

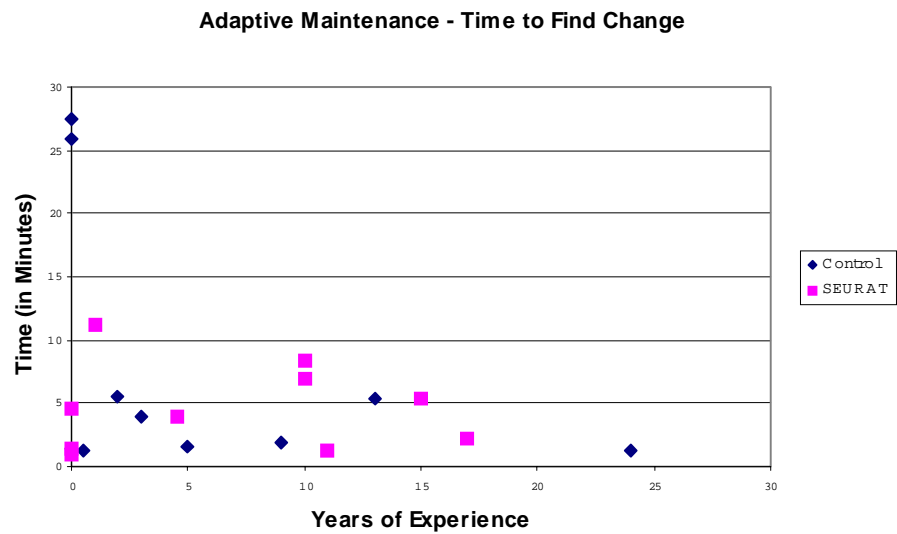


FIGURE 10-5. Adaptive Maintenance - Time to Find Change vs. Experience

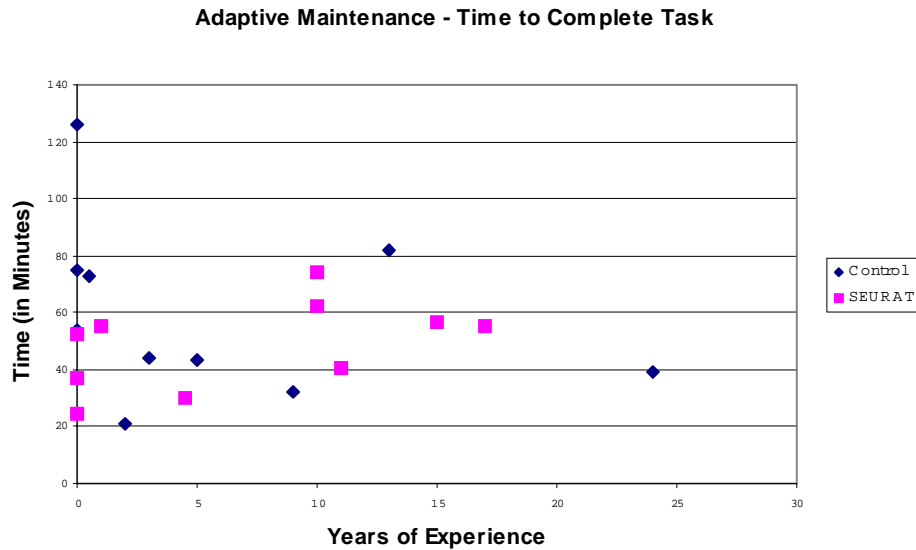


FIGURE 10-6. Adaptive Maintenance - Time to Complete Task vs. Experience

10.2.1.2. Corrective Maintenance

This was by far the simplest of the three tasks, requiring that only a single line of code be removed in order to fix the problem. Figure 10-7 shows the average time required to find the change, Figure 10-8 shows the average time to complete the task. Results are similar to those shown for the Adaptive Maintenance task except that the difference between using and not using SEURAT was very different for the group of users with only Some Java. These users had a very difficult time finding the change without the assistance of a tool.

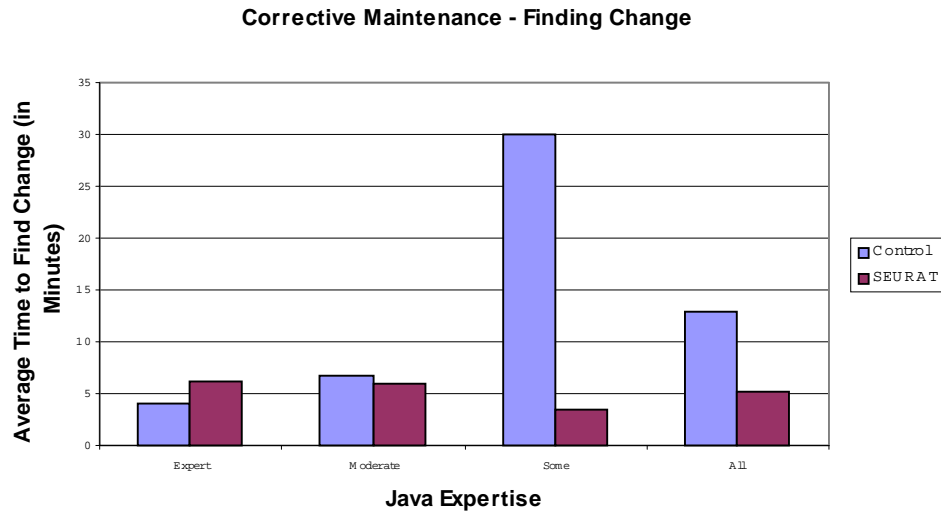


FIGURE 10-7. Corrective Maintenance - Time to Find Change

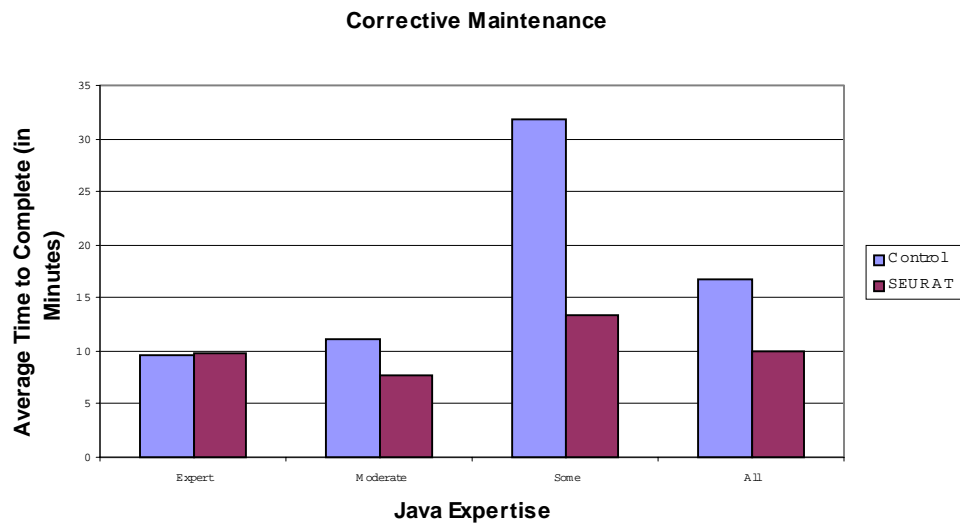


FIGURE 10-8. Corrective Maintenance - Time to Complete Task

Another interesting observation is that the variance in the times to find the change and complete the task were much larger for the control group than for the SEURAT group. Figure 10-9 shows the variance in the time to find the change while Figure 10-10 shows

the variance in the time to complete the task. The exception to this is in the group with Moderate experience – this group had more variation in the SEURAT group. Plots showing the times to find the change and complete the task are plotted versus the number of years of subject experience in Figure 10-11 and Figure 10-12.

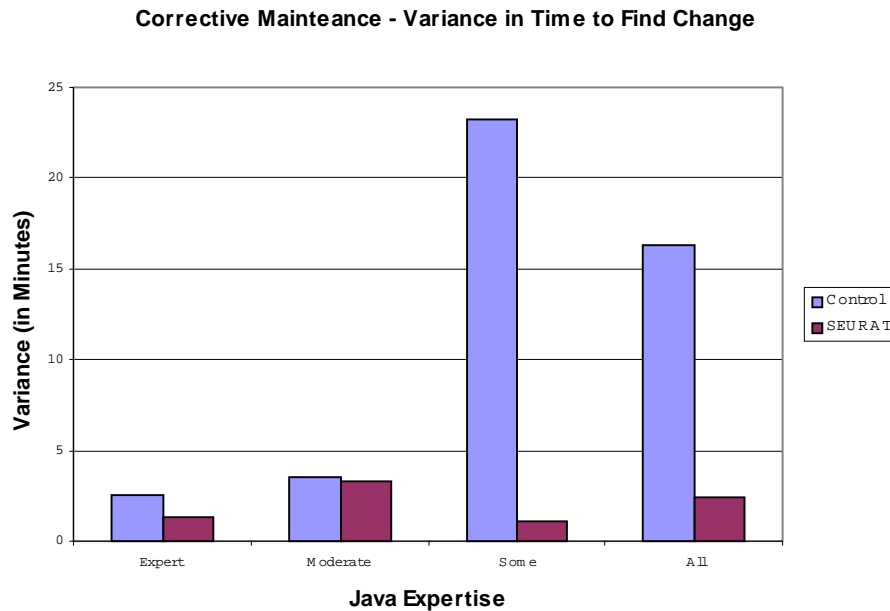


FIGURE 10-9. Corrective Maintenance - Variance in Time to Find Change

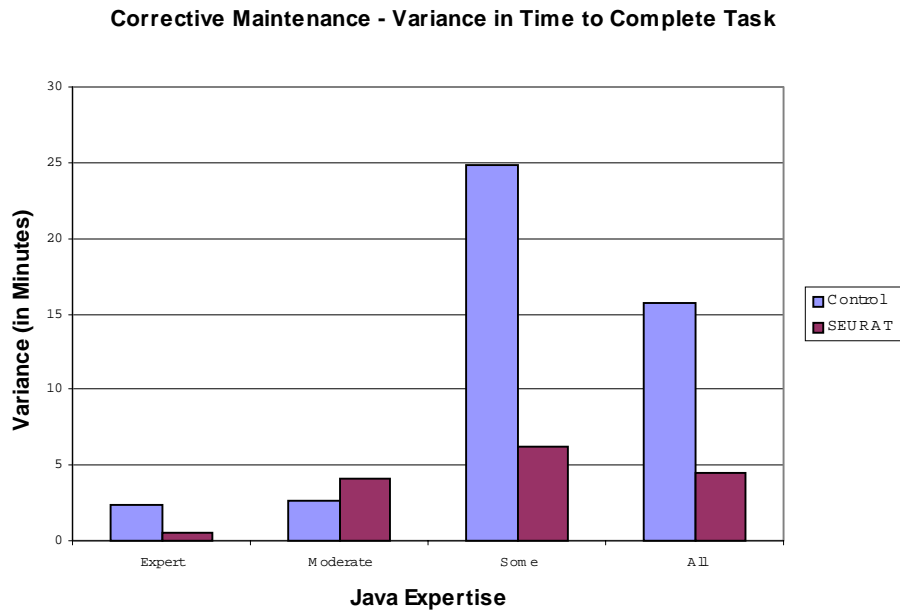


FIGURE 10-10. Corrective Maintenance - Variance in Time to Complete Task



FIGURE 10-11. Corrective Maintenance - Time to Find Change vs. Experience

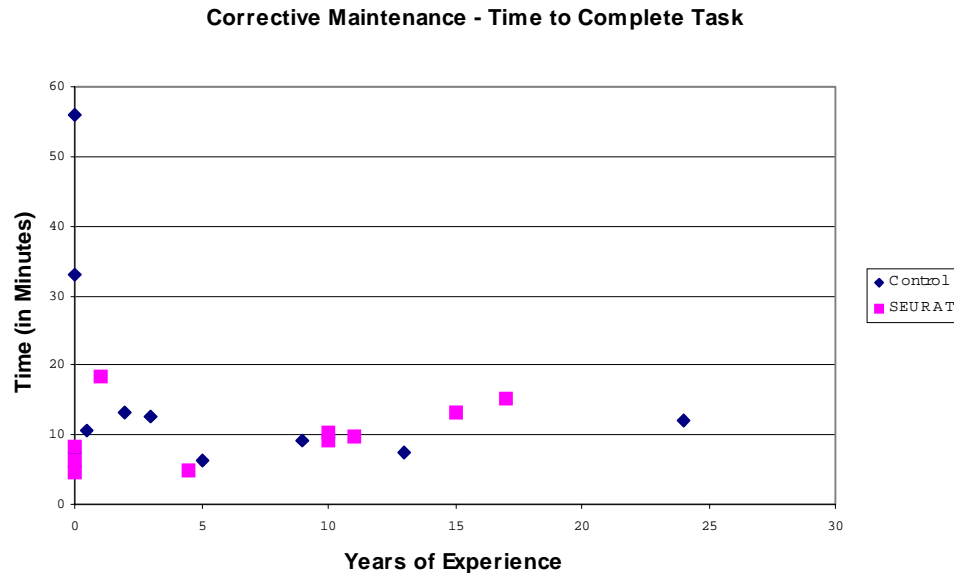


FIGURE 10-12. Corrective Maintenance - Time to Complete Task vs. Experience

10.2.1.3. Enhance Maintenance

This task was intended to be fairly simple, although not as easy as the corrective maintenance task. The intent was that the subjects would find the spot where meetings were cancelled, realize that they did not have the information needed within the method, find the call, and then realize that it was easy to pass the required information in as an argument. Instead, most of the subjects spent time trying to get the required information from within the method. Some even tried to get it from the class that originally read in the data (LoginUser) with one person going to the desperate measures of trying to read in all the user information a second time. Figure 10-13 shows the average time required to find the change, Figure 10-14 shows the average time used to complete it. The results are similar to the other experiments except that SEURAT did help the Expert user group find the change faster, although the average time to complete for experts was still faster for the control group.

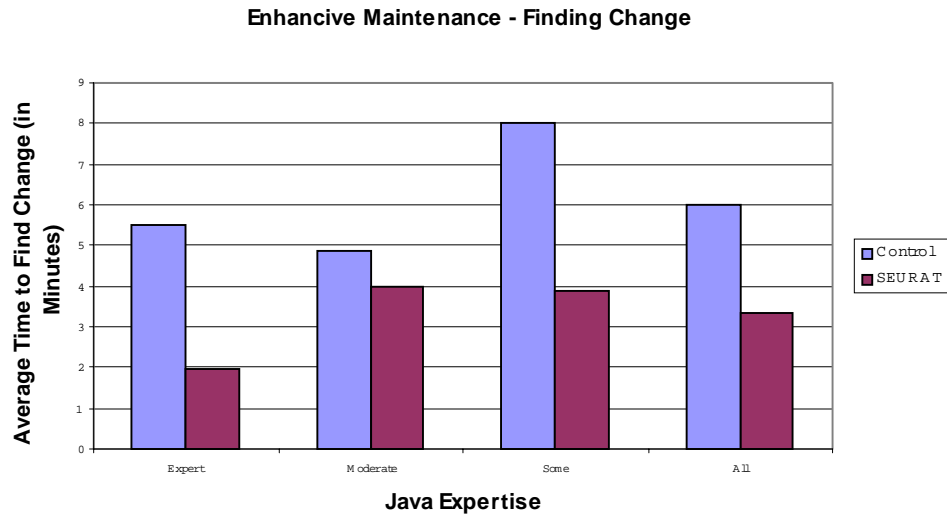


FIGURE 10-13. Enhance Maintenance - Time to Find Change

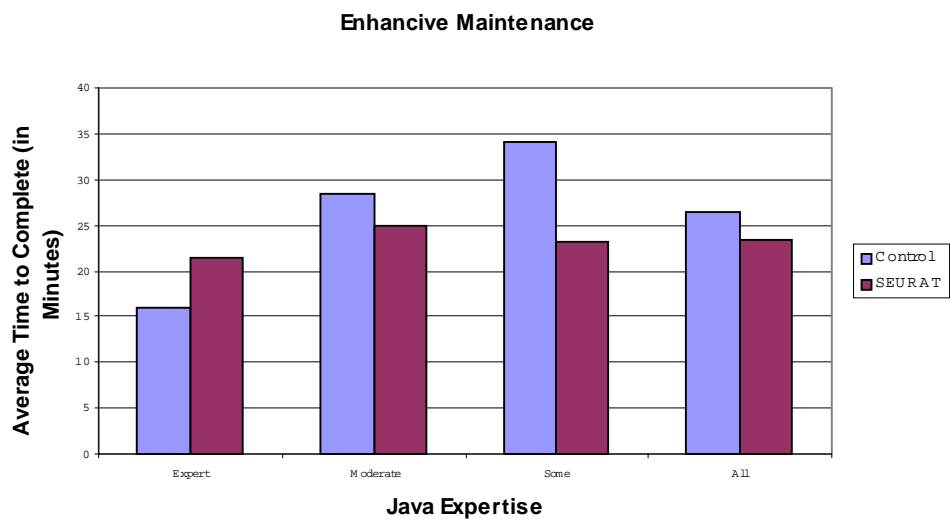


FIGURE 10-14. Enhance Maintenance - Time to Complete Task

As with the other tasks, the variance was much larger for the control group than for the group using SEURAT. Figure 10-15 shows the variance for the time required to find the code that needed to be changed and Figure 10-16 shows the variance for the time required to complete the task. Plots showing the times to find the change and complete the task are plotted versus the number of years of subject experience in Figure 10-17 and Figure 10-18.

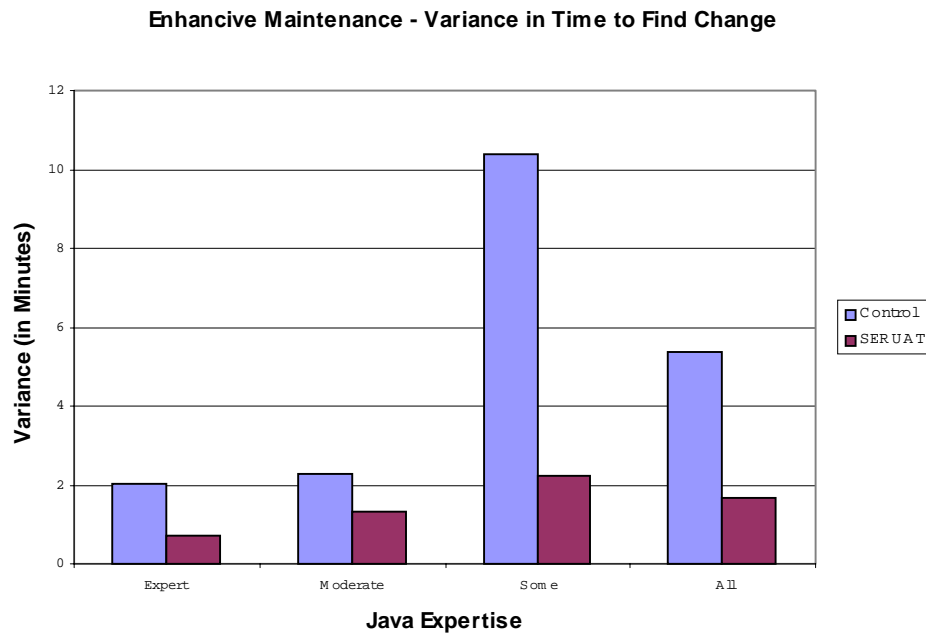


FIGURE 10-15. Enhance Maintenance - Variance in Time to Find Change

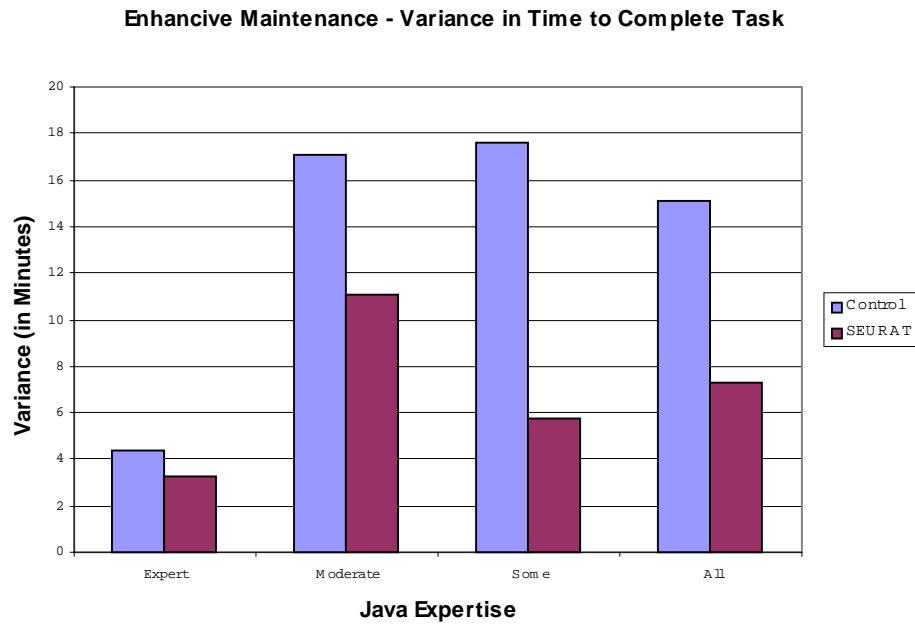


FIGURE 10-16. Enhance Maintenance -- Variance in Time to Complete Task

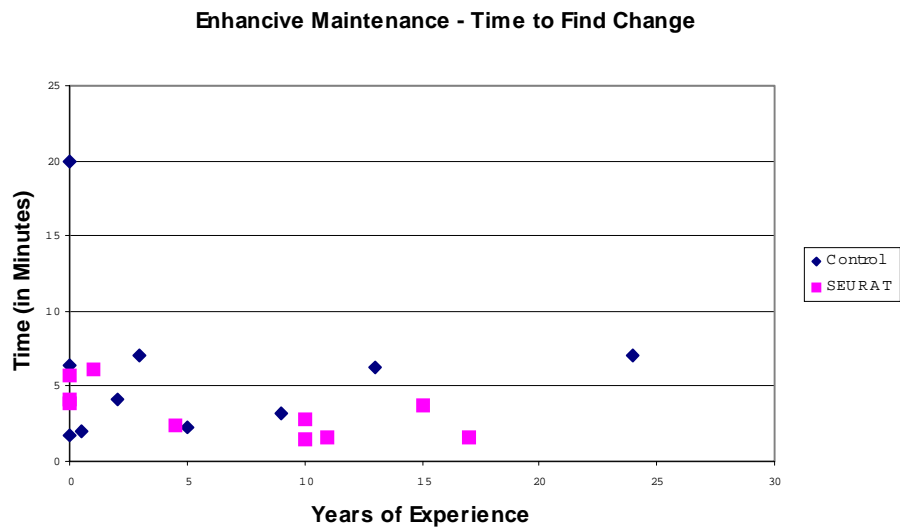


FIGURE 10-17. Enhance Maintenance - Time to Find Change vs. Experience



FIGURE 10-18. Enhancive Maintenance -- Time to Complete Task vs. Experience

10.2.1.4. Statistical Analysis

A number of different statistical techniques were used to look at the data from the three tasks to see if the improvement from SEURAT was statistically significant. The tests were performed using StatView [StatView, 1999]. The three techniques used were the F-test, a test that compares variances from two unpaired groups, the ANACOVA test, which looks at the experimental factor (use of SEURAT) and other possibly involved co-variables (the experience level and Java expertise of each subject), and the Mann-Whitney test, which like the F-test is a technique for comparing two unpaired groups but unlike the F-test does not assume a normal distribution. All tests were performed with an alpha level of 0.05 (the null hypothesis would be incorrectly rejected 5% of the time). The goal behind the statistical analysis is to show if the null hypothesis (the hypothesis that using SEURAT had no effect on the time required to complete the tasks) could be rejected.

The F-test results are shown in Table 10-4. These results show that the null hypothesis can be rejected for all of the tasks, with the strongest results being reported for the corrective maintenance task. This is indicated by the P-Value, which is the probability that the null hypothesis could be true, being less than 0.05. These results are somewhat suspect because the F-test assumes that the data follows a normal distribution and that the variances between the two groups are equal [Keppel, et. al., 1992]. The variances for the control group were typically twice that of the experimental (SEURAT) group.

TABLE 10-4. F-Test Analysis Results

	Degrees of Freedom (N=D)	F-Value	P-Value
Adaptive Delta	9	9.022	0.0031
Adaptive Total	9	4.053	0.0490
Corrective Delta	9	43.896	< .0001
Corrective Total	9	12.311	0.0009
Enhancive Delta	9	10.509	0.0017
Enhancive Total	9	4.326	0.0400

A second analysis, the ANACOVA test, was performed to compare the effect of using SEURAT while taking into account the experience of the participants and the level of Java expertise. Table 10-5 shows the results from the ANACOVA test. This test showed that only the results for the corrective maintenance task could be considered significant.

TABLE 10-5. ANACOVA Analysis Results

	Adaptive Delta	Adaptive Total	Corrective Delta	Corrective Total	Enhancive Delta	Enhancive Total
	P-Value	P-Value	P-Value	P-Value	P-Value	P-Value
Group	0.6380	0.6796	0.0004	0.0058	0.3028	0.0689
Level	0.2908	0.8340	0.0051	0.0033	0.2654	0.3556
Experience	0.0950	0.2663	0.0015	0.0019	0.1290	0.0592
Group * Level	0.8866	0.8517	0.0040	0.0511	0.4632	0.1913
Group * Experience	0.2507	0.2957	0.0015	0.0026	0.1974	0.0245
Level * Experience	0.1177	0.3030	0.0024	0.0022	0.1705	0.1212
Group * Level * Experience	0.4958	0.5522	0.0032	0.0080	0.2063	0.0465

The Mann-Whitney tests is useful because it does not assume that the data follows a normal distribution. The observations (in this case, the times) are not used, only their ranks and this makes it more resistant to outliers [StatView, 1999]. There are outliers in several of the data sets in this experiment which is another reason to consider using this analysis. Table 10-6 shows the results of the Mann-Whitney test. The P-values from this

test indicate that the null hypothesis (that SEURAT had no effect on time) could not be rejected.

TABLE 10-6. Mann-Whitney Analysis Results

	Adaptive Delta	Adaptive Total	Corrective Delta	Corrective Total	Enhancive Delta	Enhancive Total
U	47.5	43.0	38.0	37.0	28.0	49.0
U Prime	52.5	57.0	62.0	63.0	72.0	51.0
Z-Value	-0.189	-0.529	-0.907	-0.983	-1.663	-0.76
P-Value	0.8501	0.5967	0.3643	0.3258	0.0963	0.9397
Tied Z- Value	-0.189	-0.529	-0.907	-0.983	-1.664	-0.76
Tied P- Value	0.8500	0.5967	0.3643	0.3258	0.0962	0.9397
# Ties	2	0	0	0	1	0

10.2.1.5. Summary

The timing information collected during the experiment shows that the SEURAT group experts took longer to perform the tasks than the control group experts. There are a number of reasons why this could be the case:

- Differences in the skill of the different users. There was quite a bit of variation within each group that was not apparent from the survey results.
- Expert users tended to experiment more with the rationale, often looking for information they hoped would be there that was not.
- Expert users were more likely to update the rationale to select a different alternative before going to the previous alternative in the code.
- The time needed to learn SEURAT (the learning curve) had a greater effect on the expert users because it took a larger percentage of the total time needed.

Using SEURAT did result in better performance for the subjects with Moderate and Some Java experience. In particular, the users in the Control group that with only Some Java experience had a very difficult time finding where they should make the change when they did not have assistance from SEURAT. Another factor that influenced the experimental results was that some of the subjects using SEURAT took the time to update the rationale

to show that they had selected a different alternative. That resulted in a longer time required to make the change.

There were also a number of general observations from the experiment:

- The subjects were far more familiar working with requirements than decisions and often would explore the requirements before moving on to the decisions.
- Some subjects would jump into exploring code for an incorrect decision rather than looking through the entire list for the decision that applies directly to the problem they are trying to solve. Many of the pertinent decisions were further down in the list provided by SEURAT, which slowed down the time it took to find the correct one and the corresponding code.
- The subjects did not use the Rationale Tasks to assist them in finding the appropriate decisions even though several described warnings and/or errors that applied to the task they were performing. This could be because SEURAT was new to them and they did not understand how all the different parts could be useful.

10.2.2. SEURAT Usability

The ten subjects that used SEURAT filled out a survey that asked about the usability of the system. The first question asked if they felt that SEURAT was easy to use. The answers were on a Likert scale [Likert, 1932] which went from Strongly Agree (SA) to Strongly Disagree (SD), with Undecided (U) in the middle. All subjects answered A, Agree, to the question.

The next question asked what part of SEURAT was most difficult to understand/use. Seven out of ten subjects said that the rationale to code association needed improvement and that they wanted to be able to get to the code directly from the rationale rather than having to go through a bookmark. One person said that choosing the selected alternative was difficult and that they would like to see this automated more, especially the part of the selection process where they un-associated the previously selected alternative with the code. One person said it was difficult to find information of interest, one said that navigating the hierarchy was difficult, and one person said that it was difficult to tell which objects they wanted to examine in the tree and that it would be useful to have a coders vs. architects view.

The final question asked what suggestions they would have for making SEURAT easier to use. Six out of ten subjects repeated their request for better connections between rationale and code. Other suggestions included providing more training, grouping the rationale in a way to relate to the current focus, automating the de-selection of alternatives, and providing better search mechanisms.

10.2.3. SEURAT Usefulness

The survey also asked the SEURAT users to answer several questions on how useful they felt SEURAT was. Figure 10-19 shows a summary of the usefulness survey.

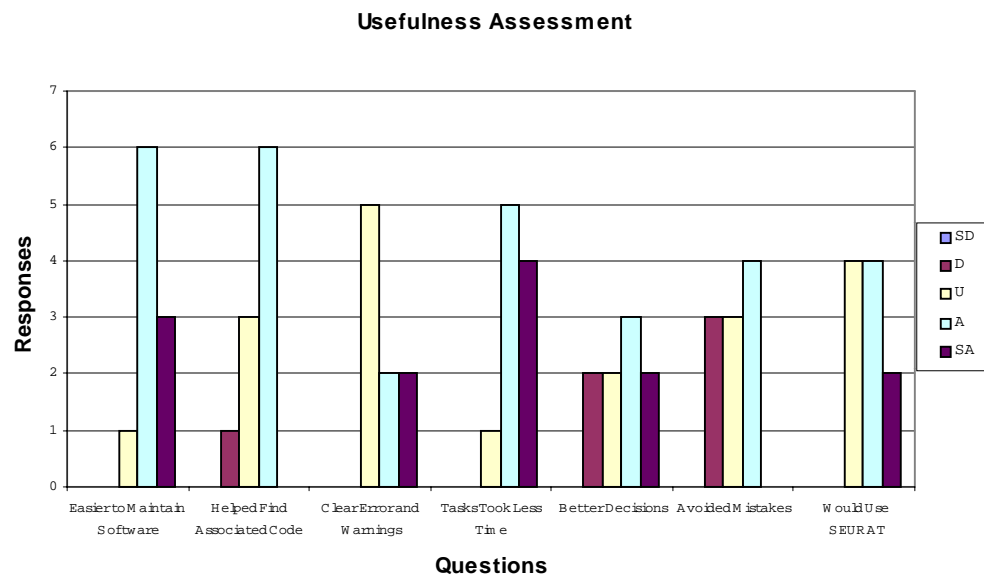


FIGURE 10-19. Usefulness Summary

Question 1: SEURAT would make it easier to maintain software

Answers: 3 SA, 6 A, 1 U

The person who was Undecided felt that the rationale given was too terse and that if it was richer then it would be more useful. The other users felt that the system would help quite a bit but one commented that it would take discipline to keep the rationale up to date.

Question 2: It was easy to find the code associated with the rationale

Answers: 6 A, 3 U, 1 D

The person who disagreed felt that the way SEURAT pulled up the code was not very intuitive and they preferred to use regular searches to look for the information. Two people who were undecided said that they had some difficulty but felt it might be because they were just learning the tool. The other person who was undecided said they had some trouble with the bookmark interface.

Question 3: The error and warning messages from SEURAT were clear and useful.

Answer: 2 SA, 2 A, 5 U, one left blank.

For the subjects answering Undecided, two did not use the errors and warnings, two did not remember seeing them (and probably did not realize this question referred to errors in the rationale, not responses to errors they might make with the tool). One said that the messages were often helpful but were difficult to understand as a novice user, and that experienced users would probably find them more useful.

Question 4: Performing the tasks took less time than they would have if SEURAT was not available.

Answer: 4 SA, 5 A, 1 U

The subject who answered undecided felt that SEURAT did help the user focus in on the problem area quickly but that after that changes were mostly driven by familiarity with the code.

Question 5: Using SEURAT helped me make better decisions.

Answer: 2 SA, 3 A, 2 U, 2 D, one blank

One subject who disagreed did not explain their answer while the other said that SEURAT helped them find the problem but that how to fix it was up to them. One of the undecided subjects did say that having the alternatives laid out might be useful but that he was not sure. The other undecided subject said that there was not much to decide but SEURAT did save them time by documenting the alternative decisions and helping them pick the best one.

Question 6: Using SEURAT helped me avoid mistakes.

Answer: 4 A, 3 U, 3 D

One subject who disagreed with this statement said that anyone could write bad code, another said that they used it mostly as a code finding tool. The other did not provide an explanation. For the undecided subjects, one felt that it did not prevent coding mistakes but did help them pick the right alternative. The other said that it helped on a high level.

Question 7: If I had SEURAT available I would use it to do my work.

Answer: 2 SA, 4 A, 4 U

Two of the undecided subjects were concerned with the overhead of maintaining the rationale. One said they would use SEURAT for large projects but not for quick prototyping. The fourth said that the usefulness would be driven by the content of the rationale.

The final question asked what features should be added (or removed) to make SEURAT more useful. Seven out of ten subjects answered this question with one or more suggestions.

The most common suggestion was to integrate SEURAT with source code – this was mentioned by three people. Two people repeated their suggestion to have direct navigation from the rationale to the source code. One person suggested that the rationale be associated with blocks of code and that SEURAT would alert the user if this code was ever changed. One person suggested linking decisions/requirements to unit tests. Integration with a UML tool was another idea. There was also a suggestion to copy SEURAT information when a java file is copied using “Save As” and one person said that the icons needed to be more readable.

10.2.4. Experiment Evaluation

Both groups were asked if the time and explanation was sufficient for all the tasks and they all agreed that it was. Some of the people in the SEURAT group felt that it was a difficult task and that one said that they were not sure how it was related to using SEURAT to improve the process.

In the control group, one person commented that the tasks would have been easier if they knew more Java. Another felt that the tasks should have been one easy, one moderate, and one difficult, rather than two that were easy and one that was difficult. Another comment was that a big part of the task was figuring out where to make the change and that would have been easier if they knew the IDE (Eclipse) or the existing code.

Three people felt that the experiment was fun – two from the SEURAT group and one from the control group. Several people in each group had no comments.

10.3. Experiment Conclusions

This experiment examined some, but not all, of the questions raised during the GQM analysis. Table 10-7 repeats the GQM analysis with a column giving results.

10.3.1. Experiment Shortcomings

The experiment was severely limited by the time constraints. As described earlier, even simple software changes can take much longer than anticipated and the experiment had to be simplified so that it could be completed in between two and three hours. Even with the simplifications, some sessions took more than three hours. The SEURAT training portion alone took thirty minutes. This was still not enough time for someone to become completely comfortable using the tool. This meant that some of the benefit of SEURAT's support may have been mitigated by the subjects needing to use the unfamiliar system.

A longer test would have been more valuable but even with the two to three hour experiment duration it was difficult to find willing subjects (even when paying them) and difficult to find time available to conduct the tests. This was particularly the case when working with subjects from industry since the tests had to be run after working hours. If more subjects had been used the results may have been more statistically significant.

The tasks were also specified in a fair amount of detail so that the subjects had an idea of what they needed to do. This meant that we could not test how SEURAT could be used to find errors in the code – this would not have been possible in the time allotted, especially for the people in the Control group because it would have required that the subjects test the system to look for defects.

The experiment also limited SEURAT use to assisting the user in finding where the code needed to be modified and, in the case of the adaptive maintenance change, suggesting a method make the change. This was helpful to the subjects but not as valuable as some of the other assistance that SEURAT could have provided that could not be readily compared in the limited time available.

TABLE 10-7. GQM Analysis with Results

Goal: To Evaluate SEURAT to determine its usefulness	Question	Metric	Result
Purpose: Evaluate Issue: using Rationale to improve the efficiency of Object (process): software maintenance Viewpoint: from the maintainer's viewpoint	Can the maintainer fix problems faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty	Faster for users with Moderate and Some Java experience. Most viewed SEURAT assistance as reducing difficulty.
	Can the maintainer perform adaptive maintenance faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty	Faster for users with Moderate and Some Java experience. Most viewed SEURAT assistance as reducing difficulty.
	Can the maintainer make enhancements to the system faster when there is rationale available to help?	Effort (person hours) or MTTR (mean time to repair) Perceived difficulty	Faster for users with Moderate and Some Java experience. Most viewed SEURAT assistance as reducing difficulty.

Purpose: Evaluate Issue: using Rationale to improve the effectiveness of Object (process): software maintenance Viewpoint: from the project manager's viewpoint	Is the maintainer better at detecting problems in the software when there is rationale available to help?	Number of problems found Time required to find problems	Not evaluated because problems were given as part of the task.
	Does the maintainer produce better solutions to problems when there is rationale available to help?	Number of errors in solutions Quality (subjective) of solutions	Longer times for some tasks indicated initial solutions did not work. Final quality not affected.
	Does the maintainer make better choices when performing adaptive maintenance when there is rationale available to help?	Number of successful adaptive improvements	Longer times for some tasks indicated that subjects tried approaches that did not initially work.
	Does the maintainer make better improvements to the system when there is rationale available to help?	Number of errors in solutions Quality (subjective) of solutions	All solutions eventually worked. Final quality not affected.
Purpose: Evaluate Issue: the usability of Object(process): the SEURAT system Viewpoint: from the maintainer's viewpoint	Is the SEURAT system easy to use?	Perceived ease of use.	All subjects felt that SEURAT was easy to use, although most had suggestions for improvement.

10.3.2. Suggested Improvements/Additional Experiments

The ideal experiment for SEURAT would be to have a long-term study of an industry project, an open source project, or possibly an academic project if one is available of

sufficient size and duration where the length of the experiment could be measured in days or weeks, not in hours. The more complex the system, the more valuable the rationale becomes and the more necessary it is to provide tool support. This would also require the collection of a rationale base to use with SEURAT. Longer term studies would also remove learning curve issues.

Constraining the experiment to comparing results found using SEURAT with a control group was a very limiting factor. There are many features of SEURAT that give the maintainer useful information that cannot be obtained with just a Java IDE. For example, SEURAT can be used to perform what-if inferencing by changing the importance of a claim or by disabling requirements and/or assumptions. Someone using SEURAT can answer the question “what decisions were made that took scalability into account” while that information is not available without the rationale. It would be useful to perform some experiments showing these features to find if they are useful to the software maintainers.

In this dissertation we described our research on using design rationale to assist with software maintenance. In it we defined rationale, described how it could assist in software maintenance, presented a system that would support these uses, and presented the results of an experiment we conducted using the system. We investigated answers to the following questions:

1. *How can rationale be used to assist in software maintenance?* We implemented a system that supported a number of uses of the rationale: presentation of relevant rationale for the software being maintained, inferencing to verify that the rationale was complete and consistent, and inferencing to check that the design decisions were well supported. Our system, SEURAT, was tightly integrated with a Java IDE so that the rationale could be captured and used from the development environment used by the developers and maintainers.
2. *How can decisions be captured with enough specificity to be useful yet still general enough to allow for inferencing?* We developed an Argument Ontology that supported reasons behind design decisions at different levels of abstraction. This provided a standard vocabulary for reasons that allowed them to be compared during inferencing. The user could also enter free text descriptions as part of the rationale.
3. *Does rationale differ for different types of software modifications?* We did not see any significant differences during our evaluation because the modifications required were very simple due to time restrictions. It is likely that the rationale will be more complicated for more complicated changes but we feel the deciding factor will be how difficult the maintenance change is and how much reasoning is required to decide to how do it, not the maintenance classification it fits into.
4. *Does maintenance rationale differ from original rationale?* The answer to this is similar to the question above. We feel the content of the rationale may be different, i.e. rea-

sons may refer to problems discovered during operation, but the structure will be the same. The maintenance changes made during the experiment did not require any structural changes to the existing rationale because they all involved selecting alternatives that were already documented in the rationale. The additional information collected concerned the reasons why different alternatives were selected which typically stated that it was because the old alternative was a poor choice and the new alternative was a better choice (i.e., no information other than that).

5. *How can rationale changes be propagated?* We performed two types of propagation through the rationale. One was during the error checking—if a change was introduced into the rationale that affected the evaluation of other alternatives and decisions, that information was propagated so that errors or warnings could be reported (or cleared) if applicable. The other form of propagation was via the importance levels set of elements in the Argument Ontology. If claims referencing the ontology had an importance value of “Default” they would inherit that importance from the corresponding ontology entry. This allows the maintainer to change the importance of that entry and propagate that change throughout the rationale. All alternatives that reference that ontology entry would be re-evaluated and warnings would be issued for cases where the alternative selected no longer had the highest evaluation.

In this last chapter we give some final conclusions about the contributions of this research (11.1) and give some recommendations for continuing this research (11.2). Finally, we summarize our findings (11.3).

11.1. Contributions

In this research, we chose to investigate how DR can be used during software maintenance and developed a system that would support those uses. The goal, in summary, was to show that with appropriate tool support, rationale can provide useful support to the software maintainer. This work contributed the following:

1. **Uses for DR during maintenance and what has to be done with the DR to support these uses:** we defined a number of ways that rationale could be useful during maintenance by helping to assure that decisions made were well supported, by verifying that requirements were not violated by the selected alternatives, by providing a way to capture dependencies between alternatives, and by providing a way to support consistent design and implementation priorities throughout the system.
2. **A method for using rationale to detect inconsistencies within the reasoning behind software decisions:** the RATSpeak representation and SEURAT system provided two ways to look for inconsistent reasoning: by inheriting argument importances using the

Argument Ontology, and by allowing the developer to explicitly define tradeoffs and co-occurrences as background knowledge.

3. A representation for rationale that supports the following:
 - a. **Rationale occurring at multiple levels in the development process from requirements through maintenance:** the RATSpeak representation allows rationale to be captured for each development phase. The Argument Ontology allows arguments to be expressed at different levels of abstraction.
 - b. **Rationale to support inferencing:** the RATSpeak representation used argumentation to support both semantic and syntactic inferencing.
 - c. **Rationale to support maintenance:** the RATSpeak representation supports both design and maintenance rationale. In addition, the reasons for making changes to the rationale are also captured.
4. **A design rationale ontology that supports inferencing by indicating the relationships between arguments at different levels of abstraction:** the Argument Ontology allows arguments to be captured in a level of detail appropriate to the stage of development and also supports the ability to compare arguments in order to evaluate the design.
5. **A way of attaching the rationale to the software implementation 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. This was done by building SEURAT as an Eclipse Plug-In so that the rationale could be associated directly with the code and so that the maintainer would be aware of when rationale was present. This integration also facilitates capture because the developer or maintainer does not need to go to a separate tool from the one they are already using to do their work.
6. **A prototype system that uses these methods to support the maintainer:** the SEURAT system allowed us to test the representation and ensure that it supports the intended uses. SEURAT was used in an experiment to evaluate the usefulness of the system and the rationale during software maintenance.
7. **Evaluation results for the prototype system:** the SEURAT system was evaluated while being used to perform three types of software maintenance: adaptive, corrective, and enhancive. The time it took to complete the tasks using SEURAT vs. not using SEURAT was compared. The subjects using SEURAT also answered survey questions about the perceived usefulness and usability of the system.

11.2. Future Work

The work with SEURAT done to date has shown a great deal of promise but there are many more areas that could be explored. These include:

-
- Expansion of SEURAT to investigate use of rationale captured at different phases of the development process;
 - Expansion of SEURAT to study multi-user rationale;
 - Longer term detailed evaluations;
 - Investigating rationale capture.

11.2.1. Investigation of Rationale for Different Phases

This would involve integrating SEURAT with additional tools used at different stages of the design process. These include requirements tools, design tools, and possibly testing tools. This would let us investigate the differences in the rationale generated and used at different stages in the development process.

This investigation would allow us to answer the two questions posed in the proposal for this work that were not addressed in this dissertation:

1. *Are there portions of the design or phases of the design process where rationale capture would be more useful than others?*
2. *What is the relationship between rationale collected at different phases?*

11.2.2. Multi-User Rationale

SEURAT is a single-user system but the MySQL database used to hold the rationale could be accessed by more than one user if it was installed on a centralized server. This could be taken a step further by associating each developer with the rationale that they enter into the system. This would greatly enhance the types of reasoning that could be used to perform evaluations.

The interactions between different developers could come in many forms:

- Developers could enter rebuttals to rationale entered by their colleagues;
- Developers could have their rationale given different values for plausibility – some users might be more knowledgeable or credible than others;
- Developers could modify rationale entered by other users.

While allowing multiple people to enter rationale does capture the spirit of argumentation, it would present some interesting challenges in evaluating alternatives when different developers and maintainers provide conflicting rationale.

11.2.3. Longer-Term SEURAT Study

The evaluation described in this dissertation was somewhat limited by having a short amount of time (2-3 hours) available with experiment subject and by being restricted to tasks that could also be performed without the tool. With shorter tasks, the benefit gained may not be surpassing the time required to learn SEURAT. The more complex the system, the more valuable the rationale becomes and the more necessary it is to provide tool support. A more realistic project would require a larger rationale base for either an industry project, an open source project, or possibly an academic project if one is available of sufficient size and duration. This would give results that would more accurately reflect the utility of SEURAT in more realistic conditions.

11.2.4. Rationale Capture

The main focus of this work has been in the use of rationale, but if more studies are to be done, there needs to be more rationale available to work with. Some tools that could be valuable sources of rationale are Configuration Management Systems and Problem Reporting Systems. Integrating SEURAT with these systems would greatly assist in capturing the rationale.

Another promising source of rationale would be code reviews and inspections. Procedures need to be developed to capture this information so that it can be made available later. There are many possible places within the software development process where rationale could be extracted. Policies, procedures, and tools need to be developed to exploit existing processes to capture the rationale.

11.3. Summary

In this dissertation, we described and demonstrated the SEURAT system, a system that integrates with a Java Interactive Development Environment to support capture and use of rationale. Building this system required developing a new rationale representation based on some of the better features of existing ones and required defining a set of useful inferences that could be performed over the rationale so that uses other than presentation could be supported. It also required developing an Argument Ontology that gives a set of reasons for choosing one design alternative over another. These reasons were arranged in a hierarchy to provide for different levels of abstraction.

SEURAT was used in a series of experiments to show how it could be used to support several different types of rationale maintenance. The results showed indications that the tool was helpful in that the average time to perform the maintenance tasks was less when

supported by SEURAT. The users with the least experience using Java enjoyed the most significant benefit.

The work performed here also has application in areas of design other than software. While the SEURAT system was designed specifically to support software development and maintenance, the rationale representation and inferencing could be used in other domains. The representation of the rationale could be easily extended to other areas of design by substituting the software-specific portions of the Argument Ontology with arguments applying to other design domains. The types of inferences implemented in SEURAT would also be applicable to domains other than software.

Rationale has many potential uses but it is perceived to be too costly and time consuming to capture. By developing a tool that provides support for capture and inferences over the rationale for compelling uses, we hope to provide additional motivation for users to make the effort to capture the rationale. The high cost of software development and maintenance has been reflected by a strong effort by many companies to improve their software development processes. This is often due to external pressure as customers expect their software to be developed at the higher levels of the Capability Maturity Model [Paulk, et. al., 1993]. Capture and use of rationale is a logical next step to continue the process improvement effort. Rationale, particularly when supported by tools such as SEURAT, has considerable promise for improving the effectiveness and efficiency of software maintenance.

References

- Araya, A., Mittal, S.: 1987, Compiling Design Plans from Descriptions of Artifacts and Problem-Solving Heuristics. *Proc. Int. Jnt. Conf. on AI, IJCAI-87*, pp. 552-558.
- Ball, L., Lambell, N., Ormerod, T., Slavin, Mariani, J.: 1999, Representing Design Rationale to Support Innovative Design Reuse: A Minimalist Approach, *from Proceedings of the 4th Annual Design Research Thinking Symposium*, MIT, May 1999, pp 1.75-1.87-
- Bañares-Alcantara, R., King, M.P., Ballinger, G.: 1995, Egide: A Design Support System for Conceptual Chemical Process Design, *AI System Support for Conceptual Design: Proc. of the 1995 Lancaster International Workshop on Engineering Design*, Springer-Verlag, New York, pp. 138-152.
- Basili, V.R., Turner, A.J.: 1975, Iterative enhancement: a practical technique for software development, *IEEE Transactions*, SE-1, (4), pp. 390-396.
- Basili, V., Caldiera, G., Rombach, G.: 1994, The Goal Question Metric Approach, *Encyclopedia of Software Engineering*, J. Marciniak (ed.), Wiley, pp 528-532
- Bennington, H.D.: 1956, Production of large computer programs, *Proc. ONR Symp. on Advanced Programming Methods for Digital Computers*, pp. 15-27.
- Bentley, P.J., Wakefield, J.P.: 1985, The Table: An Illustration of Evolutionary Design using Genetic Algorithms, *Proc. Conf. Genetic Algorithms in Engineering Systems: Innovations and Applications*, IEE Conference Publication No. 414, pp. 412-418.
- Birmingham, W.P., Brown, D.C.: 1997, IEEE Expert special issue on AI in Design, D.C. Brown and W.P. Birmingham (Guest Eds.), Volume 12, Number 2, March/April 1997, Volume 12, Number 3, May/June 1997.
- Boehm, B.: 1988, A Spiral Model of Software Development and Enhancement, *IEEE Computer*, vol.21, #5, May 1988, pp 61-72.
- Boehm, B., Bose, P.: 1994, A Collaborative Spiral Software Process Model Based on Theory W, *Third*

-
- International Conference on the Software Process*, Reston, VA, pp. 59-68.
- Bondi, A.: 2000, Characteristics of Scalability and Their Impact on Performance, *Proceedings of the 2nd International Workshop on Software and Performance*, Ottawa, Ontario, Canada, pp. 195 - 203
- Booch, G.: 1991, *Object-Oriented Design with Applications*, The Benjamin/Commings Publishing Company.
- Bose, P.: 1995, A Model for Decision Maintenance in the WinWin Collaboration Framework, *Knowledge Based Software Engineering (KBSE '95)*, pp. 105-113.
- Brandish M., Hague, M., Taleb-Bendiab, A.: 1996, M-LAP: A Machine Learning Apprentice Agent for Computer Supported Design, *AID'96 Machine Learning in Design Workshop*.
- Bratthall, L., Johansson, E., Regnel, B.: 2000, Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software Architecture Evolution, in *Proc. of the Int. Conf. on Product Focused Software Process Improvement*, Oulu, Finland, pp. 126-139.
- Brice, A., Johns, B.: 1998, Improving process design by improving the design process, QSL-9002A-WP-001, *QuantiSci*, October 1998.
- Britt, B., Glagowski, T.: 1996, Reconstructive derivational analogy: A machine learning approach to automating redesign, in *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, No.10, Cambridge University Press, pp. 115-126.
- Brooks, F.P. Jr.: 1995, *The Mythical Man-Month*, Addison Wesley
- Brown, D.: 1992, Design, in *Encyclopedia of Artificial Intelligence*, S. Shapiro (Ed.), Vol. 1, New York: John Wiley & Sons, pp. 331-339.
- Brown, D. C.: 1998, Defining Configuring, invited paper, *AI EDAM Special Issue on Configuration*, T. Darr, D. McGuinness and M. Klein (Eds.), Cambridge U.P., Vol. 12, pp. 301-305.
- Brown, D. C., Bansal, R.: 1991, Using Design History Systems for Technology Transfer, in *Computer Aided Cooperative Product Development*, D. Sriram, R. Logcher and S. Fukuda (Eds.), Lecture Notes Series, No. 492, Springer-Verlag, New York, pp. 544-559.
- Brown, D., Chandrasekaran, B.: 1989, *Design Problem Solving: Knowledge Structures and Control Strategies*, California: Morgan Kaufmann.
- Burge, J., Brown, D.C.: 2000, Inferencing Over Design Rationale, in *Artificial Intelligence in Design '00*, J. Gero (Ed.), Kluwer Academic Publishers, pp. 611-629.
- Burge, J., Brown, D.C.: 2002, *NFRs: Fact or Fiction*, Computer Science Technical Report, Worcester Polytechnic Institute, WPI-CS-TR-02-01.
- Cambell, M., Cagan, J., Kotovsky, K.: 1989, A-Design: Theory and Implementation of an Adaptive Agent-
-

-
- Based Method of Conceptual Design, in *Artificial Intelligence in Design '98*, J. Gero (Ed.), Kluwer Academic Publishers, pp. 579-598.
- Chapin, N.: 2000, Software Maintenance Types-A Fresh View, *Proc. of the International Conf. On Software Maintenance*, IEEE Computer Society Press, CA, pp. 247-252.
- Chen, A., McGinnis, B., Ullman, D., Dietterich, T.: 1990, Design History Knowledge Representation and Its Basic Computer Implementation, *The 2nd International Conference on Design Theory and Methodology*, ASME, Chicago, IL, pp. 175-185.
- Chung, L., Nixon, A., Yu, E.: 1995, Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design, *Proc. ICSE-17 Workshop on Architectures for Software Systems*, Seattle, Washington, April 24--28.
- Chung, P.W.H., Goodwin, R.: 1998, An integrated approach to representing and accessing design rationale, in *Engineering Applications of Artificial Intelligence*, 11, pp. 149-159.
- Chung, L, Nixon, BA, Yu, E, 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., Burgess-Yakemovic, K.: 1995, A Process-Oriented Approach to Design Rationale, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, (Eds.), Lawrence Erlbaum Associates, Mahwah, NJ, pp. 293-428.
- Dellen, B., Kohler, K., Maurer, F.: 1996, Integrating Software Process Models and Design Rationals, in *Proceedings Knowledge-based Software Engineering*, Syracuse, NY, IEEE Computer Society Press, pp. 84-93.
- Devanbu, P., Brachman, R., Selfridge, P., Ballard, B.: 1991, Lassie: A Knowledge-based Software Information System, *Communications of the ACM*, Vol. 34, No. 5, pp. 34-49.
- Dix, A., Finlay, J., Abowd, G. Beale, R.: 1998, *Human Computer Interaction*, 2nd Edition, Prentice Hall.
- Dixon, J.R., Howe, A., Cohen, P.R., Simmons, M.K.: 1986, Dominic I: Progress towards Domain Independence in Design by Iterative Redesign, *Proceedings of the 1986 ASME Computers in Engineering*, v. 4, pp. 199.
- Dutoit, A., Paech, B.: 2001, *Rationale management in software engineering*, S.K. Chang (Ed.), World Scientific Publishing, pp 787-816.
- Easterbrook, S.M., Nuseibeh, B.A.: 1995, Managing Inconsistencies in an Evolving Specification, *Second IEEE Symposium on Requirements Engineering*, York, UK, March 27-29, pp. 48-55.
- Filman, R. E.: 1998, Achieving Ilities, Workshop on Compositional Software Architectures, Monterey, Cal-
-

ifornia, <http://www.objs.com/workshops/ws9801/papers/paper046.doc>.

- Fischer, G., Lemke, A., McCall, R., Morch, A.: 1995, Making Argumentation Serve Design, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, (Eds.), Lawrence Erlbaum Associates, pp. 267-294.
- Frayman, F., Mittal, S.: 1987, Cossack: A Constraints-Based Expert System for Configuration Tasks, in *Knowledge-Based Expert Systems in Engineering: Planning and Design*, D. Sriram and R. A. Adey (Eds.), Computational Mechanics Publications, 143-166.
- Ganeshan R., Garrett J., Finger, S.: 1994, A framework for representing design intent, *Design Studies Journal*, V15 No. 1, January, pp. 59-84.
- Garcia, A., Howard, H., Stefik, M.: 1993, *Active Design Documents: A New Approach for Supporting Documentation in Preliminary Routine Design*, Tech. Report 82, Stanford Univ. Center for Integrated Facility Engineering, Stanford, CA.
- Garlan, D., Shaw, M.: 1993, An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering, Volume I*, V. Ambriola, G. Tortora (Eds.), World Scientific Publishing Company, New Jersey, pp. 1-39.
- Garlan, D., Monroe, R.T., Wile, D.: 1997, ACME: An Architecture Description Interchange Language, *Proceedings of CASCON '97*, Ontario Canada, pp. 169-183.
- Goel, A.: 1991, A Model-Based Approach to Case Adaptation, in *Proc. Thirteenth Annual Conference of the Cognitive Science Society*, Chicago, Hillsdale, NJ: Lawrence Erlbaum, pp. 143-148.
- Gogolla, M.: 1998, *UML for the Impatient*, Research Report 3/98, Universität Bremen.
- Gotel, O., Finkelstein, A.: 1994, An Analysis of the Requirements Traceability Problem, *1st International Conference on Requirements Engineering*, Colorado Springs, pp. 94-101.
- Gruber, T.: 1990, Model-based Explanation of Design Rationale, in *Proceedings of the AAAI-90 Explanation Workshop*, Boston, July 30, 1990.
- Gruber, T., Russell, D.: 1991, *Design Knowledge and Design Rationale: A Framework for Representation, Capture, and Use*, Knowledge Systems Laboratory, KSL 90-45, Stanford University, Stanford: CA.
- Grudin, J.: 1995, Evaluating Opportunities for Design Capture, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, (Eds.), Lawrence Erlbaum Associates, Mahwah, NJ, pp. 453-470.
- Hall, T., Rainer, A., Baddoo, N., Beecham, S.: 2001, An Empirical Study of Maintenance Issues within Process Improvement Programmes in the Software Industry, in *Proc. of the International Conference on Software Maintenance*, Florence, Italy, pp. 422-430.
- Hofmeister, C., Nord, R.L., Soni, D.: 1999, Describing software architecture with UML, in *Proceedings of*

-
- the First Working IFIP Conference on Software Architecture, San Antonio, TX, February 1999, pp. 145-159.
- Hubka, V., Eder, W.E.: 1996, *Design Science*, Springer-Verlag, London.
- Jacobson, I.: 1987, Object-oriented development in an industrial environment, *Proc. of OOPSLA'87*, Special issue of *SIGPLAN Notices* 22 (12), pp. 183-191.
- Jacobson, I., Booch, Rumbaugh, J.: 1999, *The Unified Software Development Process*, Addison-Wesley, MA, 1999.
- Jiro, K: 2000, *KJ Method: A Scientific Approach to Problem Solving*, Tokyo: Kawakita Research Institute.
- Joskowicz, L., Sacks, E.: 1999, Computer-Aided Mechanical Assembly Design Using Configuration Spaces, *IEEE Computers in Science and Engineering*, Nov./Dec., pp. 14-21.
- Kant, E.:1985 Understanding and Automating Algorithm Design. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1361-1374.
- Kajiko-Mattsson, M.:2001, The State of Documentation Practice within Corrective Maintenance, in *Proc. of the International Conference on Software Maintenance*, pp. 354-363.
- 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.
- Keppel, G., Saufley, W.H. Jr., Tokunaga, H.: 1992, *Introduction to Design & Analysis: A Student's Handbook*, 2nd Edition, W.H. Freeman and Company, New York.
- King, J.M.P., Bañares-Alcantara, R.: 1997, Extending the scope and use of design rationale records, in *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11, Cambridge University Press, pp. 155-167.
- Kitchenham, B.A., Travassos, G.H., von Mayhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., Yang, H.: 1999, Towards an ontology of software maintenance, *Journal of Software Maintenance: Research and Practice*, vol 11, pp.365-389.
- Klein, M.: 1992, DRCS: An Integrated System for Capture of Designs and Their Rationale, in *Artificial Intelligence in Design '92*, Gero, J. (ed.), Kluwer Academic Publishers, pp. 393-412.
- Klein, M.: 1997, An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture, *Concurrent Engineering Research and Applications*, March, pp. 37-46.
- Klein, M. and Kazman, R.: 1999, Attribute-Based Architectural Styles, Software Engineering Institute Technical Report CMU/SEI-99-TR-22.
- 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.
- Lee, J.: 1990, SIBYL: A qualitative design management system, in *Artificial Intelligence at MIT: Expanding Frontiers*, P.H. Winston and S. Shellard (Eds.), Cambridge MA: MIT Press, pp. 104-133.
- Lee, J.: 1997, Design Rationale Systems: Understanding the Issues, *IEEE Expert*, Vol. 12, No. 3, pp. 78-85.
- Lee, J.: 1991, Extending the Potts and Bruns Model for Recording Design Rationale, in *Proceedings of the 13th International Conference on Software Engineering*, Austin, TX, pp. 114-125.
- Lehman, M.: 2003, "Software Evolution Cause or Effect?", Stevens Award Lecture, International Conference on Software Maintenance, Amsterdam.
- Lientz, B. P., Swanson, E. B.: 1980, *Software Maintenance Management*, Addison-Wesley, Reading, MA.
- Likert R.: 1932, A technique for measuring attitudes, *Principles of Social Psychology*, N Hayes (Ed.) Laurence Earlbaum Associates, pp. 110.
- MacLean, A., Young, R.M., Bellotti, V., Moran, T.P.: 1995, Questions, Options and Criteria: Elements of Design Space Analysis, in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll (Eds.), Lawrence Erlbaum Associates, NJ, 1995, pp. 201-251.
- Madhavji, N.H.: 1991, The process cycle, *Software Engineering Journal*, 6, 5, pp. 234-242.
- Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: 1995, Specifying Distributed Software Architectures, in *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, pp. 137-153.
- Marcus, S., Stout, J., McDermott, J.: 1992, VT: An Expert Elevator Designer That Uses Knowledge-Based Backtracking, in *Artificial Intelligence in Engineering Design*, Vol. 1, C. Tong and D. Sriram (Eds.), Academic Press, 1992, pp. 317-355.
- Maurer, F.: 1996, Coordinating System Development Processes, *Proceedings Knowledge Acquisition Workshop 1996 (KAW-96)*, Banff, Vol. 2 pp. 49/1-49/20.
- McDermott, J.: 1982, R1: A Rule-based Configurer of Computer Systems, *Artificial Intelligence*, Vol. 19, North-Holland, pp. 39-88.
- Medvidovic, N., Taylor, R.N.: 1997, A Framework for Classifying and Comparing Architecture Description Languages, in *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, , Zurich, Switzerland, September 22-25, pp. 60-76.
- Mitchell, T.M., Mahadevan, S., and L. Steinberg, L.: 1985, LEAP: A Learning Apprentice for VLSI Design, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pp. 573-580.
- Mittal S., Araya, A.: 1992, A Knowledge-Based Framework for Design, in *Artificial Intelligence in Engi-*
-

-
- neering Design, Vol. 1, C. Tong and D. Sriram (Eds.) , Academic Press, pp. 273-293.
- Mittal, S., Frayman, F.: 1989, Towards a Generic Model of Configuration Tasks, *Proc. Int. Int. Conf on AI*, pp. 1395-1401.
- Mostow, J., Barley, M., Weinrich, T.: 1992, Automated Reuse of Design Plans in Bogart, in *Artificial Intelligence in Engineering Design*, Vol II, C. Tong and D. Sriram (Eds.), Academic Press, Inc. pp. 57-104.
- Murthy, S., Addanki, S.: 1987, PROMPT: An Innovative Design Tool, in *Expert Systems in Computer-Aided Design*, J. S. Gero (Ed.), North-Holland, 1987, pp. 323-341.
- Myers, K., Zumel, N., Garcia, P.: 1999, Automated Capture of Rationale for the Detailed Design Process, in *Proceedings of the Eleventh National Conference on Innovative Applications of Artificial Intelligence*, AAAI Press, Menlo Park, CA, pp. 876-883.
- Navinchandra, D., Sycara, K., Narasimhan, S.: 1991, A Transformational Approach to Case Based Synthesis, *Artificial Intelligence in Engineering, Manufacturing and Design*, Vol. 5, No. 1, May 1991, pp. 31-45.
- Niessink, F., Van Vliet, H.: 2000, Software Maintenance from a Service Perspective, *Journal of Software Maintenance: Research and Practice*, Vol. 12, No. 2, March/April, pp. 103-120.
- Nuseibeh, B., Easterbrook, S.: 2000, Requirements engineering: a roadmap, in *The Future of Software Engineering*, Special Volume published in conjunction with ICSE 2000, A. Finkelstein (Ed.), pp. 35-46.
- Nuseibeh, B., Easterbrook, S., Russo, A.: 2000, Leveraging Inconsistency in Software Development, *Computer*, Vol. 33, No. 4, IEEE Computer Society Press, pp. 24-29.
- Osterweil, L.J.: 1987, Software Processes are Software Too, in *Proceedings of ICSE 9*, Monterey, CA, pp. 2-13.
- Osterweil, L.J.: 1997, Software Processes are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9, in *Proceedings of ICSE 97*, Boston, MA, pp. 540-548.
- Paulk, M. C., Curtis, W., Chrissis, M.B., Weber, C. W.: 1993, Capability Maturity Model, Version 1.1, *IEEE Software*, Vol. 10, No. 4, July 1993, pp. 18-27.
- Peña-Mora, F., Vadhavkar, S.: 1996, Augmenting design patterns with design rationale, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11, Cambridge University Press, pp. 93-108.
- Peña-Mora, F., Sriram, D., Logcher, R.: 1995, Design Rationale for Computer-Supported Conflict Mitigation, *ASCE Journal of Computing in Civil Engineering*, pp. 57-72.
- Perry, D.E., Wolf, A.L.: 1992, Foundations for the Study of Software Architecture, *SIFSOFT Software Engi-*
-

-
- neering Notes, Vol. 17, No. 4, Oct. 1992, pp. 40-52.
- Potts, C., Bruns, G.: 1988, Recording the Reasons for Design Decisions, *Proceedings International Conference on Software Engineering*, IEEE CS Press, pp. 418-427.
- Pressman, R.S.: 1997, *Software Engineering: A Practitioner's Approach*, McGraw Hill.
- Pugh, S.: 1991, *Total Design: Integrated Methods for Successful Product Engineering*, Addison-Wesley.
- Ramachandran, N., Shah, A., Langrana, N.: 1988, Expert System Approach in Design of Mechanical Components, *Proc. Computers in Engineering Conf.*, ASME, July, pp. 1-10.
- Ramesh, B., Dhar, V.: 1994, Representing and Maintaining Process Knowledge for Large Scale Systems Development, *IEEE Expert*, Vol. 9, No. 4, pp 54-60.
- Rational: 1999, *Rational Unified Process: Best Practices for Software Development Teams*, <http://www.rational.com/products/whitepapers/100420.jsp>
- Rational: 2000, *Rational Rose 2000e: Using Rational Rose*, Rational Software Corporation 20 Maguire Rd. Lexington, MA 02421
- Reiss, S.P.: 2002, Constraining Software Evolution, in *Proc. of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, pp. 162-171.
- Robbins, J.E., Medvidovic, N., Redmiles, D.F., Rosenblum, D.S.: 1998, Integrating Architecture Description Languages with a Standard Design Method, *ICSE '98 Proceedings*, Kyoto, Japan, pp. 209-218.
- Robinson, W. N., S. Pawlowski, Volkov, S.: 1999, *Requirements Interaction Management*, GSU CISWorking Paper 99-7, Georgia State University, Atlanta, GA.
- Royce, W.W.: 1970, Managing the development of large software systems, *Proc. IEEE Wescon*, pp. 1-9.
- Rumbaugh, J., Jacobson, I., Booch, G.: 1998, *The Unified ModelingLanguage Reference Manual*, Reading, MA: Addison-Wesley.
- Runkel, J.T., Birmingham, W.P., Darr, T.P., Maxim, B.R., Tommelein, I.D.: 1992, Domain Independent Design System: Environment for Rapid Prototyping of Configuration Design Systems, in *Proc. 2nd Int. Conf. on AI in Design*, J.S. Gero (Ed.), 22-25 June, Pittsburgh, PA, Kluwer Acad. Pub., pp. 21-40.
- Rushby, J.:1994, Critical System Properties: Survey and Taxonomy, *Reliability Engineering and System Safety*, Vol 43, No. 2, pp. 189-219.
- Sim, S., Duffy, A.: 1994, A New Perspective to Design Intent and Design Rationale, in *Artificial Intelligence in Design Workshop Notes for Representing and Using Design Rationale*, 15-18 August, pp. 4-12.
- Singer, J.: 1998, Practices of Software Maintenance, in *Proc. of the International Conference on Software*
-

-
- Maintenance (ICSM'98)*, 16-19 November, 1998, Bethesda, Maryland, USA, pp. 139-145.
- Shipman, F., McCall, R.: 1996, Integrating different perspectives on design rationale: Supporting the emergence of design rationale from design communication, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11, Cambridge University Press, pp. 141-154.
- Shum, S., Hammond, N.: 1993, *Argumentation-Based Design Rationale: From Conceptual Roots to Current Use*, Tech. Report EPC-1993-106, Rank Xerox Research Centre, Cambridge.
- Siewiorik, D.: 1990, Fault Tolerance in Commercial Computers, *Computer*, Volume 23, Issue 7, pp. 26-37.
- Song, X., Leon J. Osterweil, L.J.: 1994, Engineering Software Design Processes to Guide Process Execution, in *Proc. 3rd International Conference on the Software Process*, Reston, VA, pp. 135-152.
- Stahovich, T.F.: 2001, Artificial Intelligence for Design, *Formal Engineering Design Synthesis*, E.K. Antonsson, and J. Cagan (Eds.), pp. 228-269.
- Cambridge University Press, 2001, pp. 228-269
- StatView: 1999, *StatView Reference Manual*, SAS Institute, Inc.
- Stefik, M.: 1981, Planning with Constraints (MOLGEN: Part 1), *Artificial Intelligence*, Vol. 16, No. 2, North-Holland, pp. 111-140.
- Stumpf, S., McDonnell, J.: 1999, Relating Argument to Design Problem Framing, *Proc. of 4th Design Thinking Research Symposium*, Cambridge, USA, pp. 245-253.
- Sutcliffe, A.G.: 1998, Scenario-based requirements analysis, *Requirements Engineering Journal*, 3(1), pp. 48-65.
- Sutcliffe, A.G., Ryan, M.: 1998, Experience with SCRAM, a SCenario Requirements Analysis Method, in *Proceedings of the IEEE International Symposium on Requirements Engineering: RE '98*, Colorado Springs C. Los Alamitos, CA: IEEE Computer Society Press, pp. 164-171.
- Sycara, K.P., Navinchandra, D.: 1992, Retrieval Strategies in a Case-Based Design System, in *Artificial Intelligence in Engineering Design*, Vol. 2, C. Tong and D. Sriram (Eds.), Academic Press, pp. 145-163.
- Taura, T., Kubota, A.: 1999, A Study on Engineering History Base, in *Research in Engineering Design*, Vol. 11, No. 1, pp. 45-54.
- Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J. Jr., Robbins, J.E.: 1995, A Component- and Message-Based Architectural Style for GUI Software, in *Proceedings of the Seventeenth International Conference on Software Engineering (ICSE17)*, Seattle, WA, April 23-30, pp. 295-304.
- Thompson, J., Lu, S.: 1990, Design Evolution Management: A Methodology for Representing and Utilizing Design Rationale, in *2nd International Conference on Design Theory and Methodology*, ASME, Chi-
-

-
- cago: IL, pp. 185-191.
- Tong, C., Sriram, D. (Eds.): 1992, Introduction, in *Artificial Intelligence in Engineering Design*, Vol. 1, C, Academic Press, pp. 1-53.
- Toulmin, S.: 1958, *The Uses of Argument*, Cambridge University Press, Cambridge.
- Ullman, D.:2004, An Example of Decision Management, White Paper, <http://www.robustdecisions.com/decision-management.pdf>
- Voas, J., Miller, K.: 1995, Software Testability: the New Verification, *IEEE Software*, Vol. 12, No. 3, pp. 17-28.
- Vanwelkenhuysen, J.:1995, Using DRE to Augment Generic Conceptual Design, *IEEE Expert*, Vol. 10, No. 1, pp. 50-56.
- Williams, B.C.: 1992, Interaction-Based Design: Constructing Novel Devices from First Principles, in *Intelligent Computer Aided Design*, D. C. Brown, M. Waldron and H. Yoshikawa (Eds.), Elsevier Science Publishers (North-Holland), pp. 255-274.
- Yen, J., Tiao, W.: 1997, A Systematic Tradeoff Analysis for Conflicting Imprecise Requirements, in *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, January 1997, Annapolis MD, pp. 87-96.
- Zave, P.: 1997, Classification of Research Efforts in Requirements Engineering, *ACM Computing Surveys*, Vol. 29, No. 4, pp. 315-321.
- Zozayza-Gorostiza, C. and Hendrickson, C.: 1987, An Expert System for Traffic Signal Setting Assistance, *ASCE Journal of Transportation Engineering*, 113(2), pp. 108-126.
- CLIPS Reference Manual: 1998, *Volume I: Basic Programming Guide*, Version 6.10, <http://www.ghg-corp.com/clips/download/documentation>.

This appendix presents the user's guide that was given to the experiment subjects who were using SEURAT to perform their maintenance tasks.

SEURAT User's Guide

**Janet E. Burge, David C. Brown
AI in Design Research Group
Computer Science Department
Worcester Polytechnic University
Worcester, MA 01609 USA**

15 November, 2004

For more information, contact:

jburge@cs.wpi.edu

Table of Contents

APPENDIX A	SEURAT User's Guide.....	A-1
A.1.	What is SEURAT?	A-5
A.2.	Rationale in SEURAT.....	A-6
A.2.1.	Rationale Structure	A-6
A.2.2.	Entering New Rationale	A-10
A.2.3.	Editing Existing Rationale	A-17
A.3.	The Rationale-Code Connection.....	A-17
A.3.1.	Associating Rationale with Code	A-17
A.3.2.	Finding Associated Rationale	A-18
A.3.3.	Removing Rationale Associations	A-19
A.4.	Rationale Tasks	A-20
A.5.	Rationale Queries	A-21
A.5.1.	Find Rationale Entity	A-21
A.5.2.	Find Common Arguments	A-22
A.5.3.	Find Requirements	A-23
A.5.4.	Find Status Overrides	A-24
A.5.5.	Find Importance Overrides	A-25
A.6.	Using the Rationale	A-25
A.6.1.	Modifying Importance Values	A-26
A.6.2.	Disabling Rationale Items	A-26

List of Figures

FIGURE A-1.	SEURAT Main Display	A-6
FIGURE A-2.	Relationships between rationale entities.....	A-8
FIGURE A-3.	Rationale Explorer	A-9
FIGURE A-4.	Rationale Icons	A-10
FIGURE A-5.	Requirement Editor.....	A-11
FIGURE A-6.	Decision Editor	A-12
FIGURE A-7.	Alternative Editor	A-13
FIGURE A-8.	Argument Editor	A-14
FIGURE A-9.	Claim Editor.....	A-14
FIGURE A-10.	Assumption Editor	A-15
FIGURE A-11.	Question Editor	A-15
FIGURE A-12.	Tradeoff Editor.....	A-16
FIGURE A-13.	Ontology Entry Editor	A-17
FIGURE A-14.	Package Explorer with Associations.....	A-18
FIGURE A-15.	Bookmark View	A-18
FIGURE A-16.	Alternative Showing Code Association.....	A-19
FIGURE A-17.	Rationale Task List	A-20
FIGURE A-18.	Find Entity Display	A-21
FIGURE A-19.	Select Claim Display	A-22
FIGURE A-20.	Find Common Arguments.....	A-22
FIGURE A-21.	Common Argument Display	A-23
FIGURE A-22.	Find Requirements Display	A-23
FIGURE A-23.	Addressed Requirements	A-24
FIGURE A-24.	Status Override Display	A-24
FIGURE A-25.	Importance Override Display.....	A-25

A.1. What is SEURAT?

SEURAT is an Eclipse Plug-In used to capture, display, and evaluate rationale associated with a Java project. Rationale, also known as Design Rationale (DR), is an argumentation structure that describes the decisions made while developing the software, the alternatives considered, and the arguments for and against each alternative. SEURAT can associate this information with the applicable code and perform inferencing over the rationale to look for areas where it is incomplete or inconsistent. In addition, the selected alternatives are evaluated to check to see if they are as well supported as the alternatives that were not chosen.

As the software system evolves, the rationale will grow and change. The rationale will also give new developers insight into why things are the way they are within the system. This is especially critical during software maintenance, the phase of the software life-cycle that SEURAT was designed to target.

Figure A-1 shows the SEURAT main display. There are five main parts, four of which are shown. These are:

- *Rationale Explorer* – The Rationale Explorer, shown in the upper left, gives a tree-hierarchy view of the rationale. Each item has a context-sensitive menu attached to it that allows editing and other actions to be taken.
- *Package Explorer* – This is the Eclipse standard Package Explorer, shown in the lower left. It has been enhanced so that it will display when rationale is associated with a Java file. There is an additional menu item that allows the user to remove these associations.
- *Editor Window* – This is the Eclipse editor, shown in the upper right pane. When rationale is associated with the code, it is shown in this window by the presence of a blue “i” to the left of the code. A mouse-over shows the name of the rationale.
- *Rationale Task List* – This lower right pane shows a list of rationale tasks that is analogous to the task display that shows compilation errors. For the Rationale Tasks List, each item refers to an error or warning within the rationale. Individual tasks can be overridden and not displayed.
- *Bookmarks View* – This pane shows the associations between rationale and code. It is not displayed in SEURAT Main Display because it is behind the Rationale Task List.

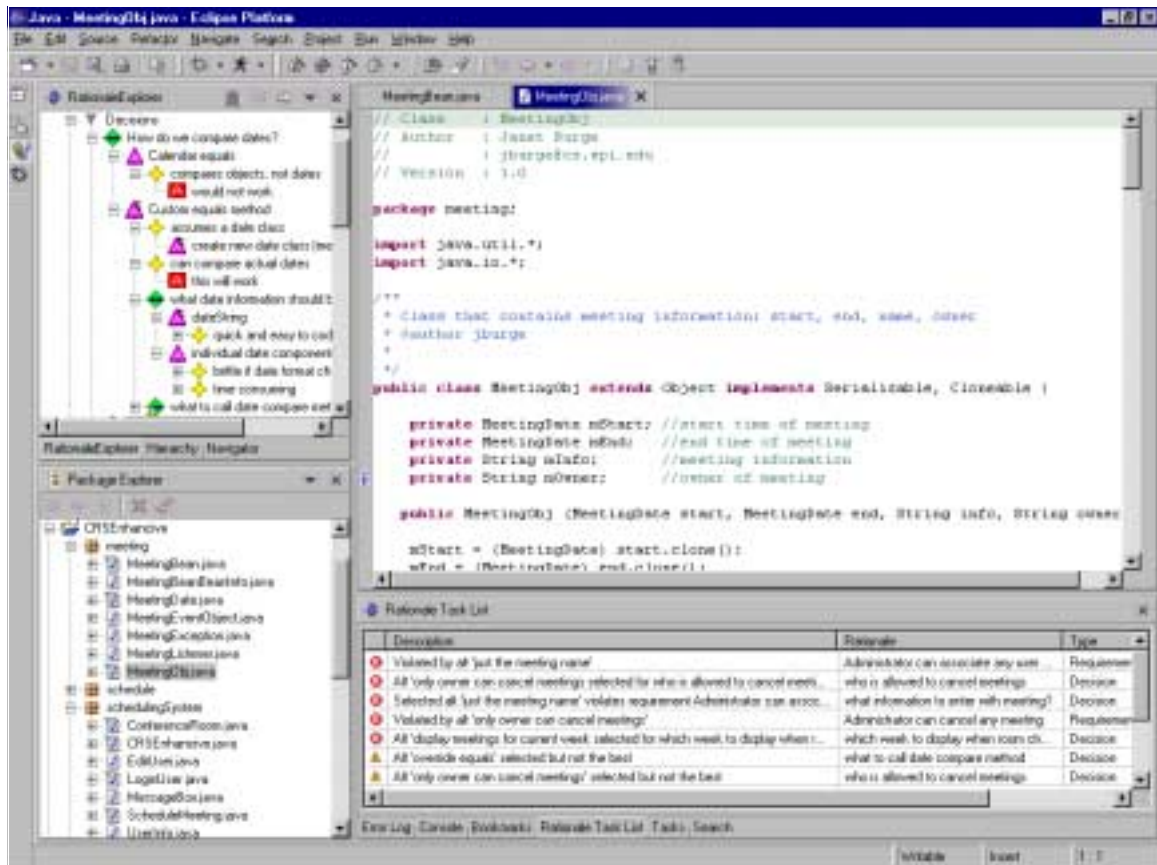


FIGURE A-1. SEURAT Main Display

A.2. Rationale in SEURAT

Rationale in SEURAT is stored as an argumentation structure. Each item of rationale contains a name as well as a more detailed description of the item. This is done to maximize the expressive power yet make it easy to process the rationale. The name and description are both stored as text strings.

A.2.1. Rationale Structure

SEURAT uses the following elements as part of the rationale:

- *Requirements* – these include both functional and non-functional requirements. They 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 requirements as part of the rationale. Requirements serve two purposes in

SEURAT. One purpose is as the basis of arguments for and against alternatives. This allows SEURAT 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. 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. A 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 vocab-

ulary 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 A-2 shows the relationships between the different rationale entities.

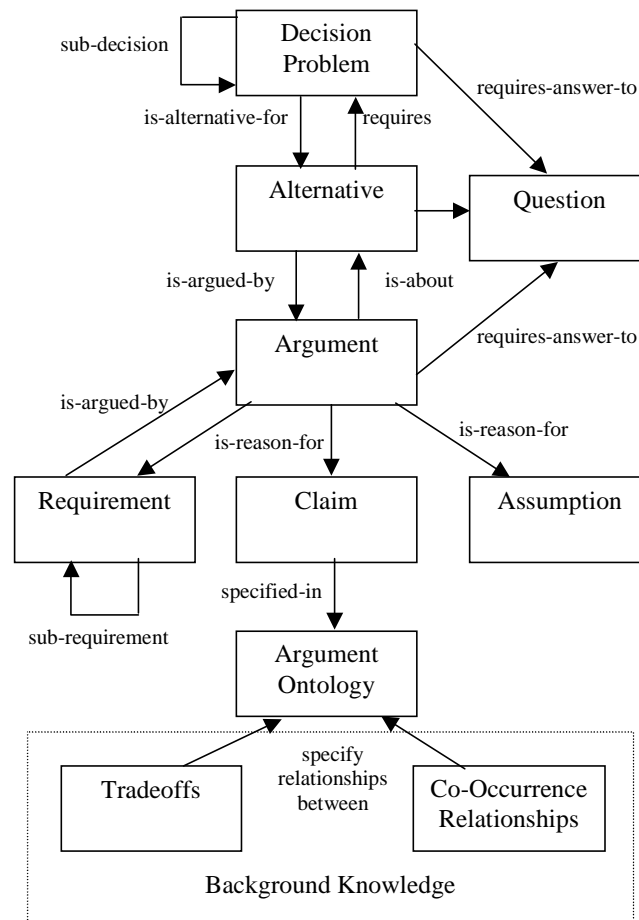


FIGURE A-2. Relationships between rationale entities

SEURAT displays the rationale in a hierarchy in the Rationale Explorer pane of the GUI as shown in Figure A-3.

Each type of rationale has its own icon as shown in Figure A-4.

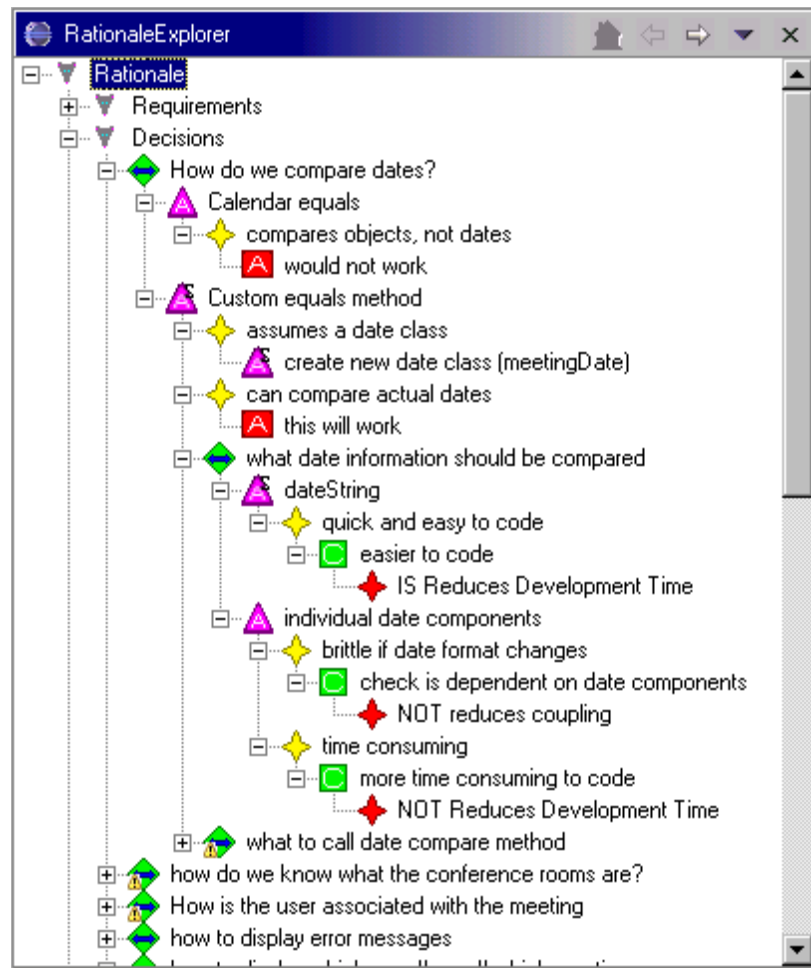


FIGURE A-3. Rationale Explorer

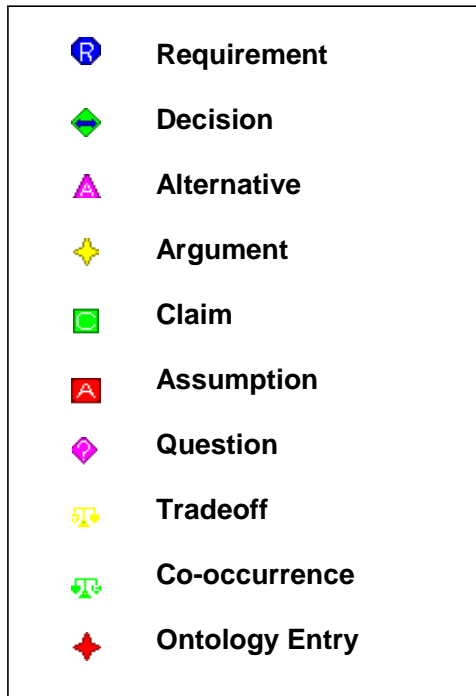


FIGURE A-4. Rationale Icons

A.2.2. Entering New Rationale

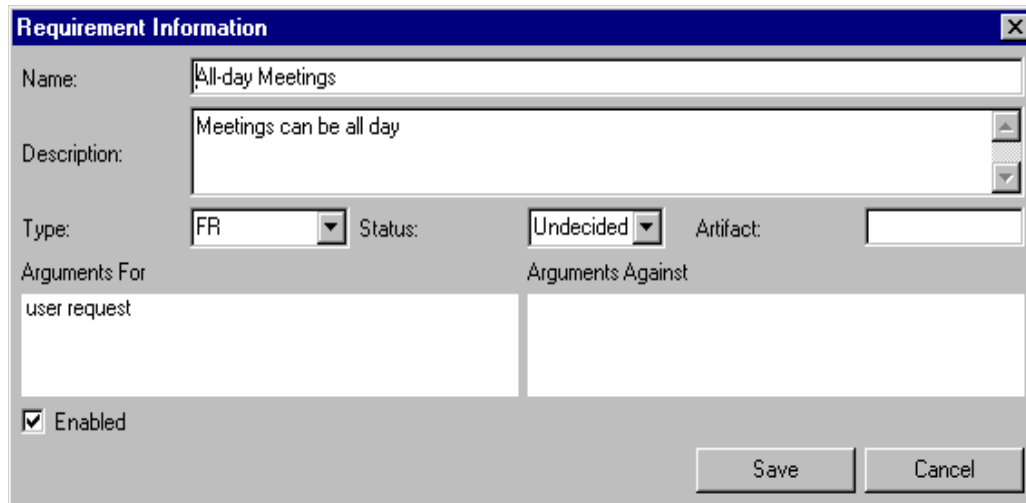
Rationale items are added via a context menu from their parent that is accessed using a right mouse click. This brings up an editor for the item that allows the user to enter the new rationale. The exceptions to this are the Claims and Assumptions, which can only be edited via the editor of a referring Argument. It is important to note that Claims and Assumptions can be shared by multiple arguments so editing one will affect the others. To find out how to check if a Claim or Argument is shared, see the Rationale Queries section of this document.

The following sections describe how to enter the different rationale elements.

A.2.2.1. Requirement

Figure A-5 shows the requirement editor. The Name is mandatory and must be filled in. Other fields will have default values. The “Arguments For” and “Arguments Against” fields are for display only and will have values if this is an existing Requirement and has arguments associated with it (as is shown here). Note the box at the lower left marked

“Enabled” this is used to disable a requirement to determine the impact on the rationale. Requirements that are not yet implemented in the current release of the software but are planned for the future can be disabled to avoid errors being displayed in the Rationale. A disabled requirement will have a “D” superimposed on its icon.



The image shows a 'Requirement Information' dialog box. It has a title bar with a close button. The fields are: Name (text box with 'All-day Meetings'), Description (text box with 'Meetings can be all day'), Type (dropdown menu with 'FR'), Status (dropdown menu with 'Undecided'), and Artifact (text box). Below these are two text boxes for 'Arguments For' (containing 'user request') and 'Arguments Against'. At the bottom left is a checked checkbox labeled 'Enabled'. At the bottom right are 'Save' and 'Cancel' buttons.

FIGURE A-5. Requirement Editor

A.2.2.2. Decision

Figure A-6 shows the Decision Editor. As with all SEURAT elements, the Name is required. There are two types of decisions: one where sub-decisions are required and one where alternatives are required. Decisions requiring sub-decisions are ones that can be broken into sub-components where answering all the sub-decision answers the parent. In this case, alternatives are not present. The example given here shows a decision that has alternatives. Each alternative is displayed here but added by right-clicking on the decision in the Rationale Explorer. Here, a numerical evaluation (rating) for the alternative is given along with its name. Higher numbers signify more support.

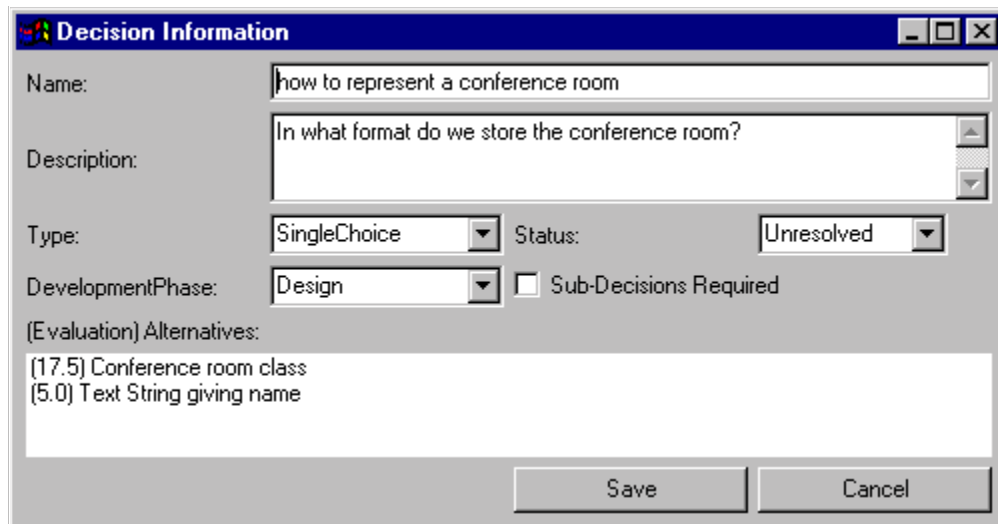


FIGURE A-6. Decision Editor

A.2.2.3. Alternative

Figure A-7 shows the Alternative Editor. This gives the information about the alternative and lists the arguments for it, against it, and that specify relationships. A relationship refers to a dependency on another alternative being selected. The Artifact field will describe what part of the code implements this alternative. In this example, the alternative has not yet been associated with any code.

A.2.2.4. Argument

Figure A-8 shows the Argument Editor. Arguments can be associated with claims (which then point into the Argument Ontology), assumptions, requirements, or other arguments. In this example, it argues a Claim, which is shown by the Argument Type field. When an argument is initially created, it is mandatory that it be associated with something. This is done using the “Select” button. When this happens, the user is allowed to either select an already existing item to use or create a new one.

Each argument gives the type, indicating if it is for or against the alternative. The possible values vary depending on the type of the argument. These are as follows:

- Claim – supports or denies
- Requirement – satisfies, addresses, or violates
- Assumption – supports or denies

-
- Argument – presupposes or opposes

In addition, the user can give the Importance of the argument, the Amount (how much the alternative meets the claim), and the Plausibility (how sure they are of the argument). The Importance can be specified as “Default,” in which case it will be inherited from the claim or Argument Ontology. Arguments involving requirements or dependencies will default to an importance of “Essential.”

Alternative Information

Name: create new date class (meetingDate)

Description: This is a sub-class of the regular calendar class.

Status: At_Issue Artifact:

Arguments For: can add new attributes/methods Arguments Against: need to create the class

Relationships

Save Cancel

FIGURE A-7. Alternative Editor

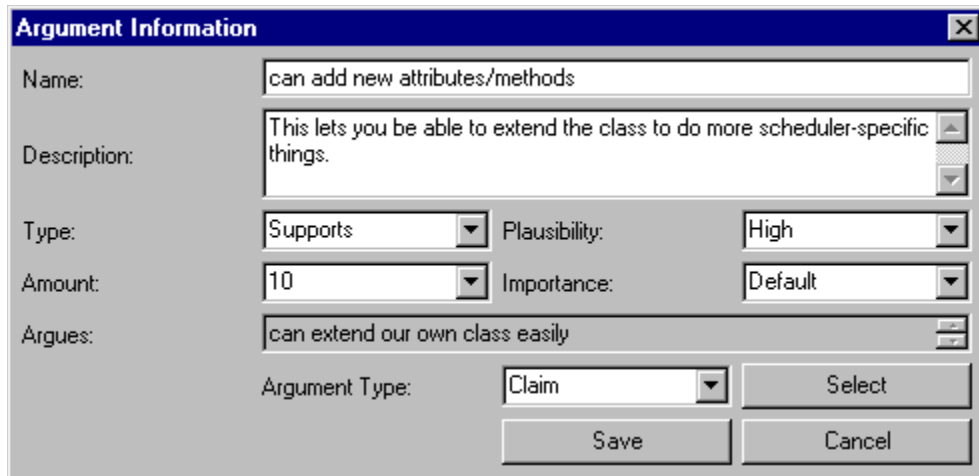
The 'Argument Information' dialog box has a title bar with a close button. It contains several fields: 'Name' with the text 'can add new attributes/methods'; 'Description' with the text 'This lets you be able to extend the class to do more scheduler-specific things.'; 'Type' with a dropdown menu set to 'Supports'; 'Plausibility' with a dropdown menu set to 'High'; 'Amount' with a dropdown menu set to '10'; 'Importance' with a dropdown menu set to 'Default'; and 'Argues' with the text 'can extend our own class easily'. At the bottom, there is an 'Argument Type' dropdown menu set to 'Claim', and three buttons: 'Select', 'Save', and 'Cancel'.

FIGURE A-8. Argument Editor

A.2.2.5. Claim

Figure A-9 shows the Claim Editor. This is similar to the Argument Editor but with fewer fields. The Direction indicates if the claim is that the alternative does what the ontology entry says, such as “IS” Reduces Development Time, or that it does not, as shown here by “NOT” Reduces Development Time. The user can also specify an importance here or inherit it from the Argument Ontology.

When a claim is created the user must associate an ontology entry with it. This is done using the “Select” button. This will bring up the ontology so the user can choose an entry to associate.

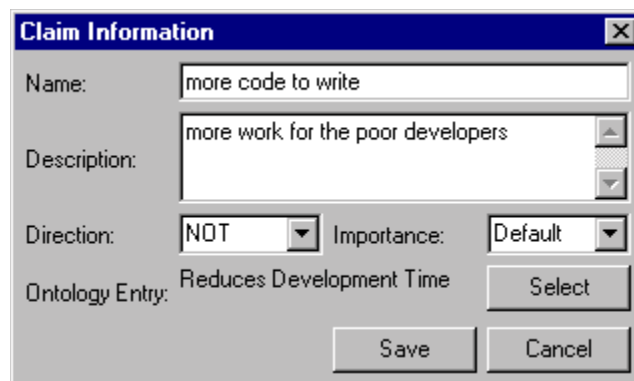
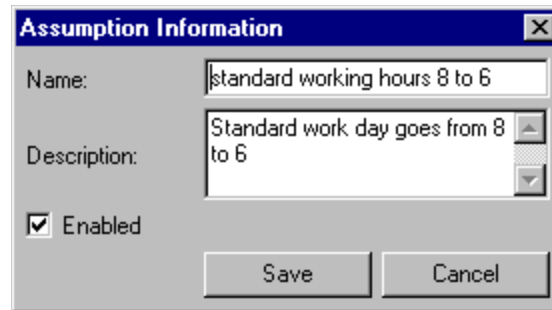
The 'Claim Information' dialog box has a title bar with a close button. It contains several fields: 'Name' with the text 'more code to write'; 'Description' with the text 'more work for the poor developers'; 'Direction' with a dropdown menu set to 'NOT'; 'Importance' with a dropdown menu set to 'Default'; and 'Ontology Entry' with the text 'Reduces Development Time'. At the bottom, there are three buttons: 'Select', 'Save', and 'Cancel'.

FIGURE A-9. Claim Editor

A.2.2.6. Assumption

Figure A-10 shows the Assumption Editor. This only requires a Name although it is more descriptive if a Description is specified as well.

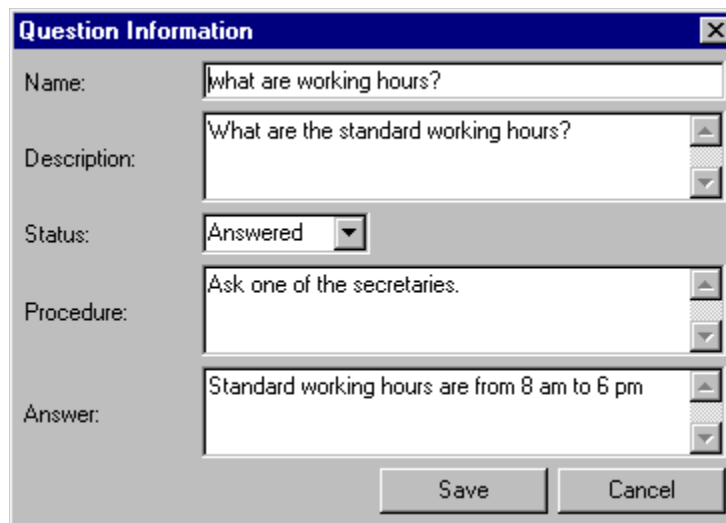


The 'Assumption Information' dialog box has a title bar with a close button. It contains three fields: 'Name' with the text 'standard working hours 8 to 6', 'Description' with the text 'Standard work day goes from 8 to 6', and a checked 'Enabled' checkbox. At the bottom are 'Save' and 'Cancel' buttons.

FIGURE A-10. Assumption Editor

A.2.2.7. Question

Figure A-11 shows the Question Editor. For each question, there is the Status that indicates if it is answered or not, a Procedure that describes the steps that must be taken to get the answer, and the Answer (once known).



The 'Question Information' dialog box has a title bar with a close button. It contains five fields: 'Name' with 'what are working hours?', 'Description' with 'What are the standard working hours?', 'Status' with a dropdown menu set to 'Answered', 'Procedure' with 'Ask one of the secretaries.', and 'Answer' with 'Standard working hours are from 8 am to 6 pm'. At the bottom are 'Save' and 'Cancel' buttons.

FIGURE A-11. Question Editor

A.2.2.8. Tradeoff

Figure A-11 shows the Tradeoff Editor. Tradeoffs are made between two Ontology Entry items. Tradeoffs can be symmetric, which indicates that they are always traded off against each other, or non-symmetric, which means the dependency is one-way. For example, in this non-symmetric tradeoff, Ontology Entry 1, Increases Flexibility, always needs to be traded off against Ontology Entry 2, Reduces Development Time. This means that if a choice increases flexibility it will increase development time. The other way around, however, is not true—if a choice increases development time it is not necessarily because of added flexibility.

The screenshot shows a dialog box titled "Tradeoff Information" with a close button (X) in the top right corner. The dialog contains the following fields and controls:

- Name:** A text field containing "flexibility vs. programming time".
- Description:** A text area containing "we know that often adding in more flexibility means more time to code and maintain." with scroll bars on the right.
- Symmetric:** A dropdown menu currently set to "No".
- Ontology Entry 1:** A text field containing "Increases Flexibility" with a "Select" button to its right.
- Ontology Entry 2:** A text field containing "Reduces Development Time" with a "Select" button to its right.
- Buttons:** "Save" and "Cancel" buttons are located at the bottom right of the dialog.

FIGURE A-12. Tradeoff Editor

A.2.2.9. Co-occurrence

The Co-occurrence Editor is identical in format to the Tradeoff Editor.

A.2.2.10. Ontology Entry

Figure A-13 shows the Ontology Entry Editor. This describes the entry and gives its Importance. This Importance will be inherited by any claims that reference the ontology entry.

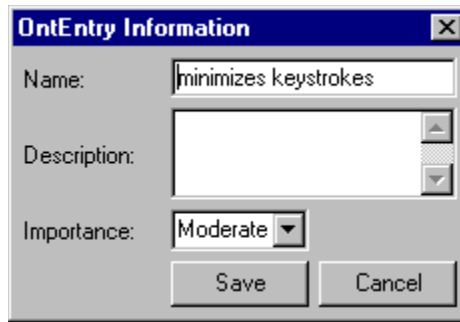


FIGURE A-13. Ontology Entry Editor

A.2.3. Editing Existing Rationale

Editing existing rationale is done by right-clicking on the Rationale Element and selecting “Edit.”

A.3. The Rationale-Code Connection

Each selected alternative should (eventually) have code that implements it. This could be a method, a class, an attribute, or a combination of several of the above. By associating alternatives with code, if the alternative needs to be re-examined it will be easier for the developer or maintainer to find where it was implemented.

A.3.1. Associating Rationale with Code

Code is associated with rationale by selecting the code in the Package Explorer and then choosing “Associate” by right-clicking on the alternative. This will display the name of the selected code item so the user can verify that this is the association they want. The icon next to the class that contains the code will then be marked with a small rat icon. Figure A-14 shows the Package Explorer where the classes MeetingDate and MeetingObj have rationale associated with them.

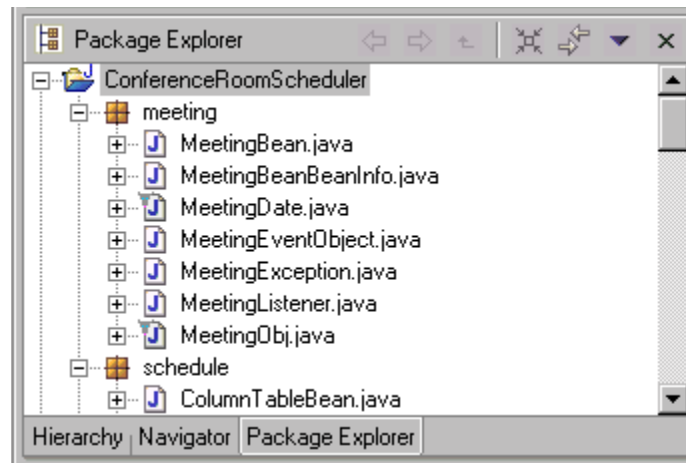


FIGURE A-14. Package Explorer with Associations

A.3.2. Finding Associated Rationale

There are several ways to find rationale associated with the code. If you have the code, and want to know if there is rationale, and what it is, the first thing to do is to look if there is a “Bookmark” in the code, denoted by an “i” in the left-hand margin. If yes, the name of the rationale can be displayed by putting the mouse over the “i.” The rationale will also be displayed on the Bookmark View. Figure A-15 shows the Bookmark View.

Bookmarks				
	Description	Resource	In Folder	Location
i	Alt: 'only owner can cancel meetings'	MeetingBe...	CRSEnhancive/meeting	line 157
i	Alt: 'override equals'	MeetingD...	CRSEnhancive/meeting	line 227
i	Alt: 'room and building combined'	Conferenc...	CRSEnhancive/scheduli...	line 20
i	Alt: 'save in a text file'	CRSEnha...	CRSEnhancive/scheduli...	line 738
i	Alt: 'select a range in a table'	CRSEnha...	CRSEnhancive/scheduli...	line 932
i	Alt: 'send individually'	MeetingBe...	CRSEnhancive/meeting	line 132
i	Alt: 'start date'	MeetingBe...	CRSEnhancive/meeting	line 198
i	Alt: 'Twenty eight day period'	CRSEnha...	CRSEnhancive/scheduli...	line 77

FIGURE A-15. Bookmark View

If the user has an alternative and they want to know the associated code, they can bring up the alternative in the editor to see if the association has been made, and to what. Figure A-16 shows an alternative with a code association listed.

The screenshot shows a dialog box titled "Alternative Information" with a close button (X) in the top right corner. The dialog contains the following fields and sections:

- Name:** A text field containing "Conference room class".
- Description:** A text area containing "Create a special class to contain the conference room information".
- Status:** A dropdown menu currently showing "Adopted".
- Artifact:** A text field containing "ConferenceRoom.java".
- Arguments For:** A text area containing "can be extended to hold more information" and "will hold the required information".
- Arguments Against:** An empty text area.
- Relationships:** A large empty text area.
- Buttons:** "Save" and "Cancel" buttons at the bottom right.

FIGURE A-16. Alternative Showing Code Association

The user can also look for the alternative in the Bookmark View and by clicking the bookmark entry, can bring up the associated code in the editor.

A.3.3. Removing Rationale Associations

Associations can be removed in two ways. The first is by right-clicking on the class in the Package Explorer and choosing "Remove Association." This will delete all associations for the file. The other, which is more selective, is by right-clicking on the bookmark in the Bookmark View and choosing "Remove Association." This will only remove the association that goes with that specific bookmark. This is useful if you do not want to remove all associations with the file. It is also possible to remove bookmarks by choosing

“Delete” but that will not remove the association even though it will remove it from the display.

A.4. Rationale Tasks

When problems are detected in the rationale, they are displayed in the Rationale Task List. The icon next to the problem indicates if it is an error or a warning. The task states what the problem is, what rationale item it is associated with, and the type of the rationale item. Right-clicking on the task and selecting “View” will bring the rationale up in the editor. If the problem is one that the user wants to ignore, they can override the task by selecting “Override” when right-clicking on the task. This will not delete the task, it will just stop displaying it. Figure A-17 shows the Rationale Task List.

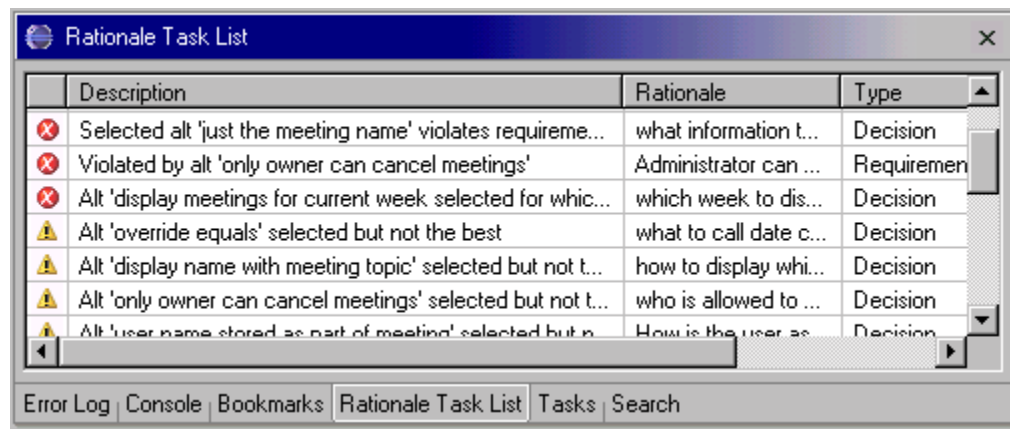


FIGURE A-17. Rationale Task List

The following errors and warnings are detected and reported by SEURAT:

- Error: No alternative is selected for a decision.
- Error: A requirement is violated.
- Error: A tradeoff is contradicted by having normally opposing arguments on the same side of the argument.
- Error: A co-occurrence is contradicted by having items appearing on opposing sides of the argument rather than together.
- Error: An alternative pre-supposes another alternative that is not selected.
- Error: An alternative is opposed by another alternative that is selected.
- Error: An alternative is selected that has arguments opposing it but none supporting it.

-
- Error: An alternative has contradictory arguments (the same, or similar, argument before and against).
 - Error: More than one alternative has been selected for a decision when only one is allowed.
 - Error: A decision requires sub-decisions to be decided and the sub-decisions are missing.
 - Warning: The alternative selected is not as well supported as other choices.
 - Warning: The alternative selected has no arguments supporting it.
 - Warning: A tradeoff is violated by missing an element
 - Warning: A co-occurrence is violated by missing an element.
 - Warning: A question has not been answered.
 - Warning: An alternative has duplicate arguments.

A.5. Rationale Queries

At the top of the Rationale Explorer there is a downward arrow that allows the user to bring down a menu of query options (the Rationale Query Menu). These are described in the following sections.

A.5.1. Find Rationale Entity

The Find Rationale Entity option allows the user to search for particular types of rationale entities (requirements, decisions, alternatives, etc.). The user is first instructed to specify the type of entity as shown in Figure A-18.

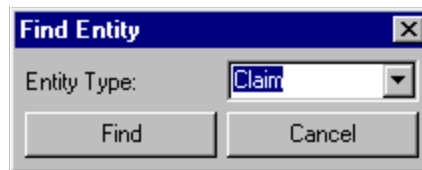


FIGURE A-18. Find Entity Display

This then brings up a list of items of that type, as shown in Figure A-19. The user can search for all or part of the item name to find it in the list. After find it, the user then can bring up the item in an editor by using the “Edit” button. The user can also choose to expand it in the hierarchy shown in the Rationale Explorer by using the “Show” button. This is helpful if the user wants to know the context around the entity.

A.5.2. Find Common Arguments

Another useful query is to find out what are the most common arguments are. This can be done for each of the three types: argument, claim, and ontology entry. Selecting which type is the first step, as shown in Figure A-20. The user can also indicate if they are only interested in common arguments for selected alternatives.

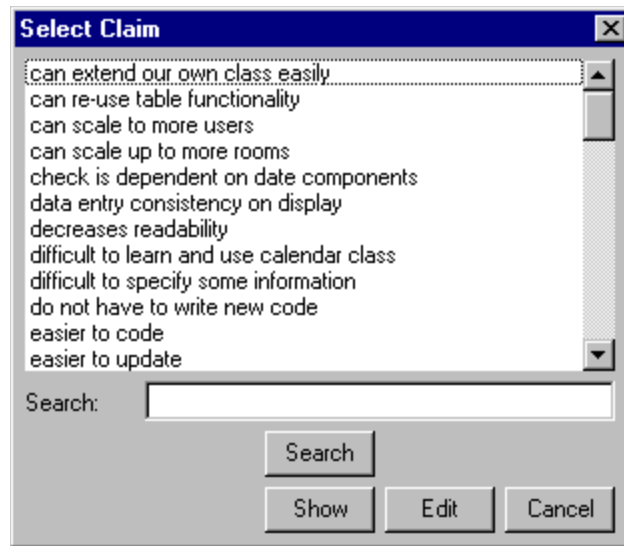


FIGURE A-19. Select Claim Display

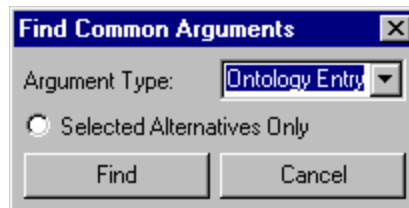
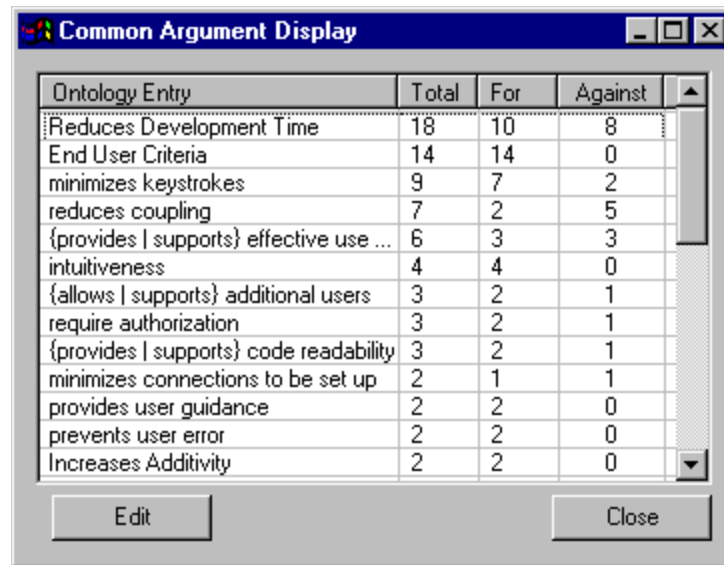


FIGURE A-20. Find Common Arguments

After selecting the type, the arguments are then displayed in a table giving the total references, the number of times it was used to argue in support for an alternative and the number of times it opposed an alternative. Figure A-21 shows the Common Argument Display showing ontology entries.

A.5.3. Find Requirements

The Find Requirements query lets the user look for requirements by their status. For example, they could get a list of all the violated requirements or all the satisfied requirements. The user selects the type using the display shown in Figure A-22. The list of requirements is shown in Figure A-23.



The image shows a dialog box titled "Common Argument Display". It contains a table with four columns: "Ontology Entry", "Total", "For", and "Against". The table lists various requirements and their counts. Below the table are two buttons: "Edit" and "Close".

Ontology Entry	Total	For	Against
Reduces Development Time	18	10	8
End User Criteria	14	14	0
minimizes keystrokes	9	7	2
reduces coupling	7	2	5
{provides supports} effective use ...	6	3	3
intuitiveness	4	4	0
{allows supports} additional users	3	2	1
require authorization	3	2	1
{provides supports} code readability	3	2	1
minimizes connections to be set up	2	1	1
provides user guidance	2	2	0
prevents user error	2	2	0
Increases Additivity	2	2	0

FIGURE A-21. Common Argument Display

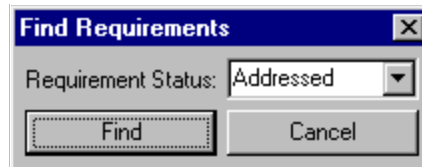


FIGURE A-22. Find Requirements Display

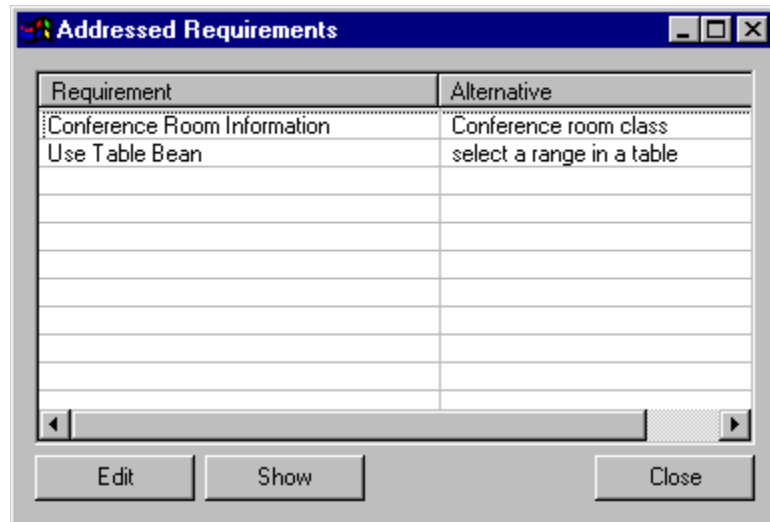


FIGURE A-23. Addressed Requirements

A.5.4. Find Status Overrides

The user can choose to override any of the items given in the Rationale Task List. This will keep the error or warning from being displayed in the list or indicated by an error or warning icon in the Rationale Explorer. The list of overridden items can be shown by choosing “Find Status Overrides” in the Rationale Query menu. Figure A-24 shows the Status Overrides display. The user can remove any override from this list and the Rationale Task List and Rationale Explorer will be updated when they exit from the display.

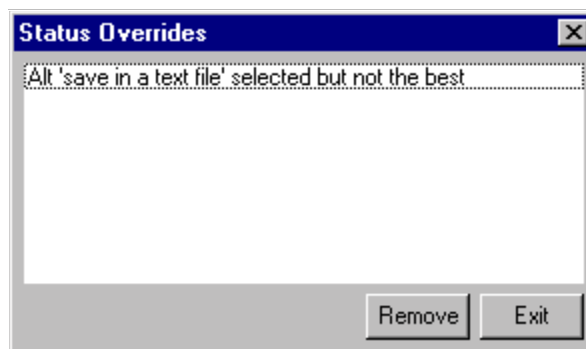


FIGURE A-24. Status Override Display

A.5.5. Find Importance Overrides

The user can also display a list of all claims and arguments where the default importance has been overridden. This can happen in several ways:

- A claim (which could be used by many arguments) has been given an importance other than the default inherited from the Argument Ontology.
- An argument that refers to a claim has been given an importance other than the default inherited from the claim (which inherits from the Argument Ontology)
- An argument that refers to a requirement has been given an importance of something other than “Essential.”

This display is brought up by selecting “Find Importance Overrides” from the Rationale Query Menu. Figure A-25 shows the Importance Override Display.

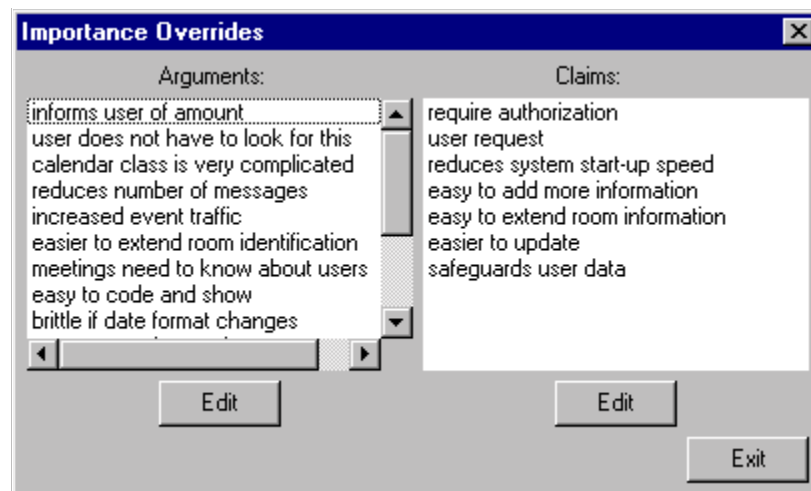


FIGURE A-25. Importance Override Display

A.6. Using the Rationale

There are a number of interesting things that can be done using SEURAT and the rationale. Two of these deserve special mention because they can be very useful in performing software maintenance.

A.6.1. Modifying Importance Values

One thing that can help find places where decisions should be re-visited is to modify the importance values and see if that results in any selected alternatives being less supported than the non-selected alternatives. For example, it might be interesting to reduce the importance of claims that refer to alternatives as being easy to code or reducing development cost. This could point out alternatives that were chosen because they were easier to code up than others. Conversely, making an item in the Argument Ontology more important could find places where modification might be needed. For example, scalability could be increased in importance to see if there were choices made that were less scalable than their alternatives.

A.6.2. Disabling Rationale Items

There are two places where disabling an item would be useful. One is with the assumptions made when developing a system. If the assumption no longer holds true in the future, those decisions should be revisited. SEURAT can be used to find where the code might need to be modified. The other useful place is with requirements. If a requirement no longer is valid then it is good to know what alternatives were chosen in order to meet the requirement and what alternatives were rejected because they violated the requirement.

This appendix presents two types of surveys used in the experiment. The first is an initial survey given to all subjects prior to the experiment. This asks general information about their experience and was used to divide them into the two experiment groups. The second type is the post-experiment survey. There are two versions: a survey for the experimental group that asks them about SEURAT usability, SEURAT usefulness, and some general questions about the experiment and a survey for the control group that asks general questions about the experiment.

Initial Survey – SEURAT Evaluation

Please return to Janet Burge's mail box by 5 pm, November 3.

The purpose of this survey is to get an idea of your background to aid in analyzing experimental results. It is important that this information be accurate.

Name: _____

e-mail (so I can contact you if schedules change!): _____

If you've worked in the software industry, for how long (this should be actual time spent – time in school between internships does not count):

College Degrees received and year:

BS: Area: _____ Year: _____

MS: Area: _____ Year: _____

PhD: Area: _____ Year: _____

Any degrees in progress? _____

Level of Java Experience (circle one): None Some Moderate Expert

Please describe your last three Java projects:

1.

2.

3.

Have you used Eclipse before? (circle one): Yes No

What are the dates/dates/times when you are likely to be available to do the experiment (or if it's easier, what dates/times are bad – make sure you indicate which you are putting down!)? All experiments will be performed at WPI in Worcester or CRA in Cambridge.

I understand that the results of this experiment will be published and give my permission for this to happen under the condition that this will be done anonymously.

I would like to be listed as a contributor in Janet's dissertation (circle one):

Yes No

Post-Experiment Survey - Experimental Group

Name: _____

Date: _____

Most questions in this survey follow a Likert scale where a statement is made and your level of agreement is one of the following:

SA: strongly agree
A: agree
U: undecided
D: disagree
SD: strongly disagree

Questions of this type will consist of a statement, followed by a choice of options. Please circle the one that best applies.

Part I: SEURAT Usability

1. SEURAT was easy to use.

SA A U D SD

2. What part of SEURAT was the most difficult to understand/use?

3. What suggestions would you have for making SEURAT easier to use?

Part II: Usefulness of SEURAT

1. SEURAT would make it easier to maintain software

SA A U D SD

Explain:

2. It was easy to find the code associated with the rationale

SA A U D SD

Explain:

3. The error and warning messages from SEURAT were clear and useful.

SA A U D SD

Explain:

-
4. Performing the tasks took less time than they would have if SEURAT was not available.

SA A U D SD

Explain:

5. Using SEURAT helped me make better decisions.

SA A U D SD

Explain:

6. Using SERUAT helped me avoid making mistakes.

SA A U D SD

Explain:

7. If I had SEURAT available I would use it to do my work.

SA A U D SD

Explain:

8. What features should be added (or removed!) that would make SEURAT more useful?

Part III: Experiment Evaluation

Quality of Explanation

1. Sufficient instruction was given to me to understand how SEURAT works

SA A U D SD

2. The explanation of task 1 was sufficient for me to understand what I needed to do

SA A U D SD

3. The explanation of task 2 was sufficient for me to understand what I needed to do

SA A U D SD

4. The explanation of task 3 was sufficient for me to understand what I needed to do

SA A U D SD

Time Allowed (if applicable)

5. Sufficient time was allowed to complete task 1

SA A U D SD

6. Sufficient time was allowed to complete task 2

SA A U D SD

7. Sufficient time was allowed to complete task 3

SA A U D SD

8. Any additional comments on the experiment?

Post-Experiment Survey - Control Group

Name or ID: _____

Date: _____

Most questions in this survey follow a Likert scale where a statement is made and your agreement is one of the following:

SA: strongly agree

A: agree

U: undecided

D: disagree

SD: strongly disagree

Questions of this type will consist of a statement, followed by a choice of options. Please circle the one that best applies.

Experiment Evaluation

Quality of Explanation

1. The explanation of task 1 was sufficient for me to understand what I needed to do

SA A U D SD

2. The explanation of task 2 was sufficient for me to understand what I needed to do

SA A U D SD

3. The explanation of task 3 was sufficient for me to understand what I needed to do

SA A U D SD

Time Allowed (if applicable)

4. Sufficient time was allowed to complete task 1

SA A U D SD

5. Sufficient time was allowed to complete task 2

SA A U D SD

6. Sufficient time was allowed to complete task 3

SA A U D SD

7. Any additional comments on the experiment?
