

Using Rationale to Support Pattern-Based Architectural Design

Wei Wang
Miami University
Benton Hall, Oxford Ohio 45056 USA
+1-513-529-0347
wangw2@muohio.edu

Janet E. Burge
Miami University
Department of English
Benton Hall, Oxford Ohio 45056 USA
+1-513-529-0347
burgeje@muohio.edu

ABSTRACT

Architectural design rationale describes the decisions made, alternatives considered, and reasons for and against each alternative considered when defining a software architecture. At least some of these reasons should reference the non-functional requirements (NFRs) for the system. The SEURAT_Architecture system uses a pre-defined pattern library and the NFRs for a software system to guide the selection of architectural patterns. Each pattern recommended by the system serves as an alternative to the architectural decision made and comes with rationale for why this pattern is considered useful. This system serves several purposes—to guide the architect through the decision-making process, to ensure that NFRs are considered when making these critical early decisions, and to capture the rationale for the architecture as a byproduct of the tool-supported selection process.

Categories and Subject Descriptors

D.2.11 [Architecture]: Patterns

General Terms

Documentation, Design

Keywords

Architectural Design, Design Patterns, Architectural Patterns, Design Rationale

1. INTRODUCTION

A Software Architecture is the structure of the components of a system, their quality attributes and the relationships among them [4]. It specifies the big picture of the potential system and lays the foundation for the next steps of the development process. Architecture design is also the first step towards addressing how the design will meet the system's requirements.

Typically, the architecture design artifacts only record the final form of the software architecture, such as the components and connectors involved. The knowledge and information behind the design decisions is often lost. If available, this background

information, a key component in architectural knowledge, can be used to help with justification of design outcomes and support for software architecture design and maintenance activities [43]. Another benefit from this architectural knowledge is to support the verification of the achievement of quality attributes. Quality attributes, sometimes referred to as Non-Functional Requirements (NFRs) [29], are important to the success of the system. The fulfillment of NFRs should be addressed early in the development cycle and built into the result of architectural design. However, in actual software projects consideration of NFRs is often delayed to later stages. This is mainly due to the lack of techniques to support incorporating NFRs in the early phases of software architecture [30].

There has been an increasing interest and growing recognition of the importance in capturing and managing architectural knowledge in the software architecture community [1][5]. This type of information is called architectural design rationale. Architecture design rationale captures the background information and reasoning process of a software architecture design, including the requirements that motivate the design, the negotiation process that leads to the final shape of the design, tradeoffs made, and alternatives that are considered. Architecture design rationale is considered an important part of software architecture and can help designers and architects with their design and maintenance activities.

In this paper, we describe an approach where NFRs are used to drive the architectural design process by assisting with the selection of architectural patterns. This approach, integrated into the SEURAT Rationale Management System [8][10], also supports the process of capturing architectural design rationale.

The remainder of this paper is organized as follows. Section 2 gives background information on the areas of design rationale and architectural patterns and describes related work. The proposed approach is presented in Section 3. Section 4 describes an example that applies the approach to a design problem. Finally, Section 5 draws conclusions and describes plans for future work.

2. BACKGROUND

Design rationale captures the background information and reasoning process of software architecture design. Some early research on the importance of the design rationale has been done [40]. However, design rationale is usually not captured and used in practice. Horner and Atwood [26] describe the inherent limitations to developing systems that can effectively capture and use design rationale. Recording the reasoning process of design can be very time-consuming and expensive. There have

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SHARK '10, May 2-8, 2010, Cape Town, South Africa
Copyright © 2010 ACM 978-1-60558-967-1/10/05 ... \$10.00

been several approaches or models that have been proposed to capture design rationale. Some argumentation-based approaches include Issue-Based Information Systems (IBIS) [31], Procedural Hierarchy of Issues (PHI) [37], Questions, Options, and Criteria (QOC) [35], Decision Representation Language (DRL) [32], RATSpeak [8], and the WinWin approach [7]. Design rationale can be used in different ways, such as helping with the verification of the satisfaction of requirements and the determination of the implication of modifications [10]. Dutoit, et al. [20] discuss how to use design rationale to support collaboration in design teams and improve quality. Burge and Brown [9] describe how design rationale can support software maintenance and ensure that reasoning given for modifications is consistent with the designer's initial intent.

The importance of architectural rationale has been recognized in the software architecture community [1][5]. The reason why considerable effort has been spent on architectural knowledge is the lack of quality documentation methodologies for the reasoning process. The focus of architectural design rationale is on treating software architecting as a process of decision making and the software architecture as a set of design decisions [28][27][8][17]. The architects operate as the decision makers during the architecting process rather than as only architecture modelers. The resulting architecture is the result of a set of design decisions during the reasoning process, such as structuring decisions and deployment decisions [44]. At the same time, there has been an attempt to capture, record, and represent the architectural knowledge/architectural rationale of the reasoning process so that the information can be used to consistently deliver a satisfactory architecture design and support achieving the quality attributes [1][20][42]. Also, models or templates of documenting software architecture and design decisions have been proposed [13][47].

Software systems are designed not only to implement all the required functionality, but also to fulfill some non-functional aspects, such as reliability, modifiability, maintainability, security. These non-functional aspects are treated as Non-Functional Requirements (NFRs). NFRs are important to the success of the software systems [18]. Ineffectively dealing with NFRs can lead to serious software system failures. Although it is difficult and expensive to deal with these requirements, the fulfillment of NFRs should be addressed early in the development cycle and built into the result of architectural design [16]. Some frameworks have been proposed to specify the relationships between NFRs and design decisions, such as [14] and [45]. NFRs can also be used as design drivers to support architectural design [15][25].

Architectural patterns specify the solutions to problems that occur over and over again in specific contexts during the software design phase [12]. Patterns provide effective approaches to developing software with defined properties. Architectural patterns are composed of a series of recurring decisions made by software design experts and also describe rationale behind these decisions, so that reference can be made on them to help with decision-making during the design process [46]. Sometimes identifying the decisions to make is already a problem for the designers because the systems to be built are too complicated to find out what to do. Identifying the right decisions to make can direct and make sure the design is along the process that trends to the successful design. Patterns can be used to direct and help the process of generating the final

architecture [12]. In [6], the authors describe how to use patterns to derive architectures. At the same time, the design decisions which have been made and why they were made this way can be recorded [39]. This knowledge will make it easier to make changes to the architecture when needed. Patterns are not isolated [23]. There are some relationships between some of the patterns. A simple catalog-like list of patterns cannot express the relationships, thereby providing the need for a pattern system/pattern language [12]. A pattern system organizes individual patterns together with the description of how patterns are connected with each other in the system, how these patterns can be implemented, and how the software development can be supported with patterns. The authors of [12] define the requirements to build a pattern system and build up a pattern system example. Avgeriou and Zdun [3] also propose a pattern language, which acts as a super set of existing architectural pattern collections and categorizations to establish the relationships between the patterns and design a categorization based on the concept of "architectural views". The PAKME knowledge management tool [2] captures patterns in their architectural knowledge repository and makes them available for reuse. The pattern library used in SEURAT_Architecture is an extension of that used in PAKME.

Harrison and Avgeriou [25] propose an approach similar to ours in their Pattern-Driven Architectural Partitioning approach, which uses functional and non-functional requirements to drive pattern selection and evaluation. This approach proposes combining patterns, which ours does not, but does not go as far into the design process or attach rationale.

3. INTEGRATING PATTERNS AND RATIONALE

In order to solve the problem of how to help architects make design decisions, and record the rationale behind these decisions, a new system has been built. First, there is a uniform method to describe the architectural patterns. This description not only specifies the pattern itself, but also exposes the relationships with other patterns and the rationale that can be used to help make the decision to choose this pattern. Second, there is an approach to organize and classify all the patterns so that the users can quickly identify which pattern or patterns can help them with their design activities. Third, there is a guideline to show how to apply or implement the patterns. Fourth, there is an evolution mechanism in the system so that it supports the changes to the existing patterns or adding new patterns to the system. The system described here supports the following functionality:

- Setting up a basic design process to help the architects to identify the key design decisions to make;
- Providing candidate patterns for the decision problems to help with the decision making. The later candidate patterns are influenced by the previous decisions made by the architects;
- Recording the decisions made and the reasons associated with them. Pre-stored common rationale for selecting a particular pattern are generated automatically and the architects can also create their own reasons for the decisions.

These goals were achieved by extending the existing Software Engineering Using RATIONale (SEURAT) system to create SEURAT_Architecture.

3.1 Software Engineering Using RATIONALE (SEURAT)

The SEURAT system [8] is a plug-in to the Eclipse IDE of the Eclipse Foundation (www.eclipse.org). It is used to capture, display, and maintain design rationale. SEURAT captures design rationale with the RATSpeak representation language, which is based on DRL [32]. RATSpeak uses requirements (both functional and non-functional) as part of the arguments and uses an Argument Ontology to give common arguments. The Argument Ontology contains 277 reasons for making design decisions starting with the general “ilities” (reliability, scalability, usability, etc.) and forming a hierarchy where the reasons become more and more specific. These ontology entries appear in several places in the rationale: associated with NFRs, associated with claims (which capture desirable or undesirable qualities that are not necessarily required), and associated with known design tradeoffs stored as background knowledge. The SEURAT system presents the rationale in an argumentation-based hierarchical structure that is displayed in an Eclipse View called the Rationale Explorer. The user can enter the decision problems, the alternatives considered for each of the decision problem, and the arguments for and against each alternative. SEURAT also supports evaluation of alternatives and performs inference over the rationale to detect inconsistency or incompleteness, such as whether a selected alternative is the best solution for the design problem or if rationale appears to be missing.

SEURAT system is designed to support the use of rationale in software maintenance. This is achieved by presenting the relevant design rationale as needed and allowing entry of new rationale for modifications [9]. The new rationale will be checked for inconsistencies with the original rationale.

3.2 SEURAT_Architecture

SEURAT_Architecture is built on the basis of SEURAT as a new feature. As an Eclipse plug-in, SEURAT_Architecture adds a new Eclipse View—a Pattern Library that includes a description of each design pattern and its rationale. SEURAT_Architecture follows the argumentation-based representation of rationale of SEURAT and supports the architectural design by supporting searching for candidate patterns, automatic generation of alternatives and arguments, and automatic generation of decisions required to implement an adopted alternative, when available in the Pattern Library.

3.2.1 Pattern Library

Patterns provide assistance in building systems with desired quality attributes. They capture proven design solutions and knowledge of some recurring problems under specific contexts. They are developed on the basis of successful design experiences of expert designers. Therefore, other designs can benefit from learning the solutions in patterns that experts use and adopt the solutions into their own projects. However, just cataloging a list of patterns cannot fully reflect the potential benefits from them. First, there are a large number of patterns out there. It is difficult to be totally familiar with each of them, especially for those “non-expert” designers. For instance, in the widely referenced design pattern book written by the Gang of Four (GoF) [24], there are 23 patterns. There are many other architectural patterns and idioms for detailed design besides those patterns in the Gang of Four book. Therefore, when a design deals with a design problem, it is time-consuming to go

through each of them. Second, patterns are not isolated [23]. The interdependencies between patterns can also benefit the designs. For example, some patterns can be better used by combination with other patterns. Third, the decisions made and rationale of the decisions during applying a pattern are easily omitted subconsciously from documentation. Sometimes some rationale in a pattern is not specified explicitly, or the reason for adopting the pattern can be considered very obvious (by the architect). In these situations, the background knowledge is lost. The information can be useful in the future software development activities, such as system maintenance.

Therefore, in our approach, a Pattern Library is designed and implemented to resolve the issues mentioned above. The Pattern Library divides the patterns into different categories so that the searching scope can be narrowed down according to the categories and the problem to be solved. The Pattern Library explicitly represents the quality attributes which each pattern influences. These quality attributes can be used as rationale during the argumentation process when a pattern is selected as a decision solution. The Pattern Library also connects patterns that are related, so that when one pattern is applied related patterns can also be considered.

There are four key components to the Pattern Library implemented for SEURAT_Architecture:

- Pattern Categories
- Design Problem Categories
- Affected Quality Attributes
- Decisions required to adopt a pattern and their Candidate (alternative) Patterns

It is difficult for the designers to read, analyze, and understand every pattern to find out the one useful because there are a lot of patterns out there and the number is still increasing. Therefore, it is helpful to divide the patterns into different categories. Here the pattern categories in [12] are adopted. Patterns are distinguished into three main categories: Architectural Patterns, Design Patterns, and Idioms.

Architectural Patterns are used at the beginning of design to solve basic fundamental structure issue of the systems. They specify the decomposition of the system, the responsibilities of subsystems, and the rules and guidelines to manage the relationships between them.

Design Patterns are used for more detailed design when refining the architecture of a system or some local design aspects of subsystems. Design Patterns address smaller design problems than architectural patterns, but are more general than the idioms, which are programming language specific.

Idioms are low-level patterns, which are used to solve the implementation of programming language specific issues, such as memory management in C++.

Every pattern is aiming at addressing a particular problem. Problem categories divide the patterns into groups according to the design situations. The problem categories from [12] are adopted for the Pattern Library:

- *From Mud to Structure* – patterns that support decomposing a system task into subtasks;
- *Distributed Systems* – patterns for systems where patterns are distributed across different processes or in several subsystems;

- *Interactive Systems* – patterns for systems that interact with humans;
- *Adaptable Systems* – patterns that provide infrastructures to support extension and adaptation;
- *Structural Decomposition* – patterns that support decomposition of complex systems;
- *Organization of Work* – patterns that define how components work together;
- *Access Control* – patterns that guard and control access to other services or components;
- *Management* – patterns to control collections of objects, services and components;
- *Communication* – patterns for organizing communication between services or components;
- *Resource Handling* – patterns to manage shared objects or components.

There are patterns that do not fall into any of these problem categories and there are patterns that can be categorized into more than one category. It is not necessary to allocate patterns into problem categories, but problem categories can help to narrow down the scope of searching for patterns.

For each of the patterns, in order to apply the pattern to a design problem there are a series of decisions that need to be made. These decisions, or decision-problems, are explicitly represented in the Pattern Library. Once a pattern gets adopted as the

solution in SEURAT, the decisions associated with it are imported into the SEURAT Rationale Explorer as decisions that need to be made. The user can apply the pattern to the design by selecting (adopting) an alternative (solution) for the decision problems. In some cases, these solution alternatives are other patterns. In this case, these patterns are imported into SEURAT as alternative solutions to the decision problems, along with the rationale why the pattern should or should not be chosen as an alternative.

For example, one of the decisions required to apply a Three-Layer pattern is “What is the structure of business logic layer?”. There are some patterns can used to solve this problem, such as Transaction Script pattern, Domain Model pattern, and the Table Module pattern. These patterns serve as alternative solutions for the business logic layer structure decision and each has rationale for why it should (or should not be) chosen. These patterns can be associated as Candidate Patterns for the decisions in the Pattern Library. If the architect selects the Three Layer pattern as an alternative, SEURAT_Architecture imports the decisions required to apply it, along with the alternative solutions for each decision and their rationale.

The system architect can add or edit patterns in the Pattern Library. This supports the evolution of SEURAT_Architecture as patterns are added or modified. Figure 1 shows the Pattern Library and an example pattern.

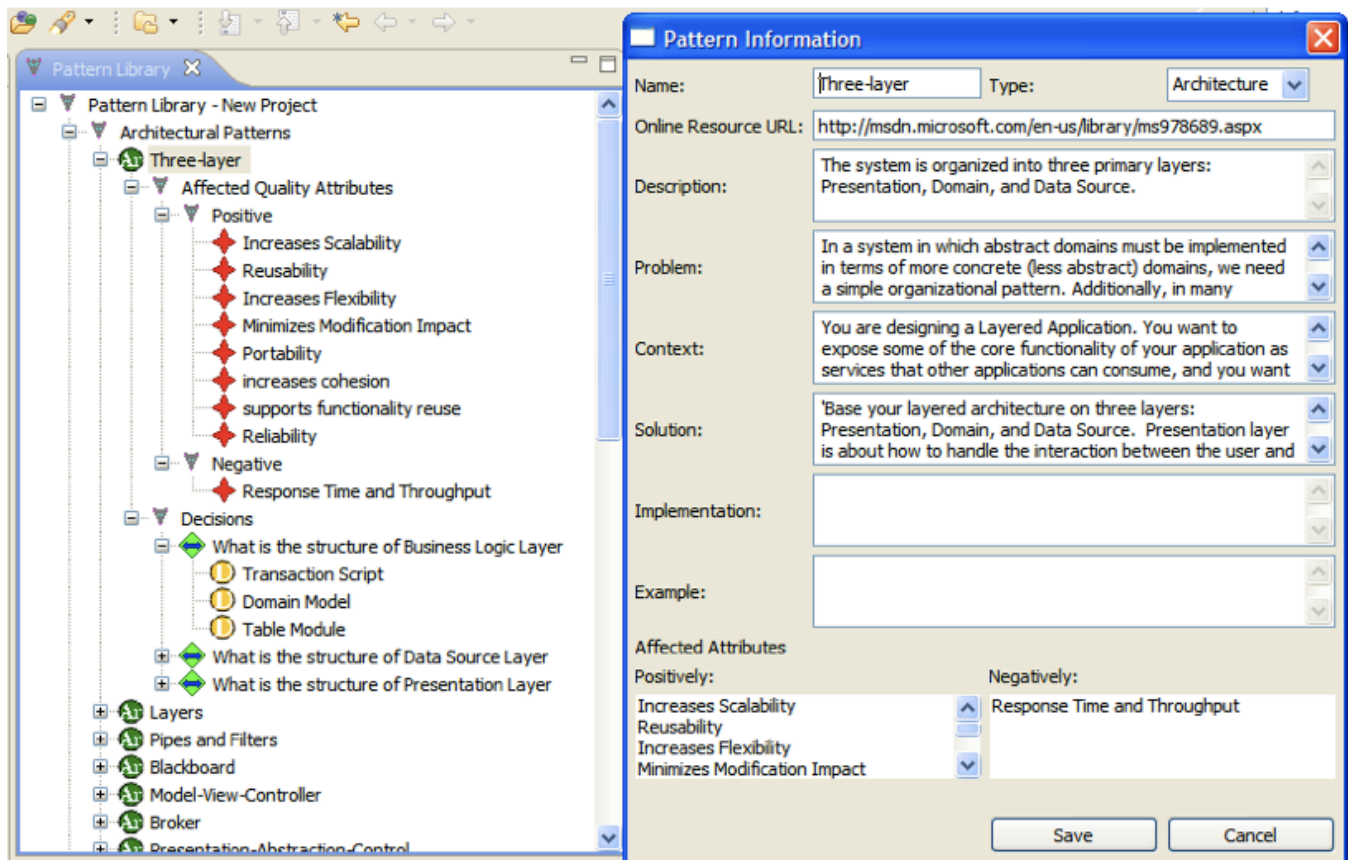


Figure 1. Pattern Library

3.2.2 Architectural Rationale Generation

SEURAT_Architecture guides the architectural design process using NFRs that the software architect enters into the system. Each NFR will be associated with an Ontology Entry in the Argument Ontology. For instance, one NFR for our example is “The software must fail no more than once per month of normal operation.” This NFR can be associated with the Ontology Entry “Availability”. The association between the NFR and Ontology Entry standardizes each NFR with a common vocabulary of quality attributes. It lays out a basis for the matching between NFRs and patterns.

NFRs and Patterns can be matched on the basis of their respective associations with Ontology Entries. If they are matched, we can say the NFR can be achieved by adoption of the matched pattern. We have two types of methods to match NFR and Pattern: Exact Matching and Contribution Matching.

In the Exact Matching method, if one NFR is associated with an Ontology Entry that is also associated with a pattern, we say the pattern is matched. For example, an NFR is described as “The main components of the software must be reusable” and associated with the Ontology Entry “Reusability”. The Three-Layer pattern affects reusability positively, so it is associated with “Reusability”. In this case, we say the Three-Layer pattern is matched with this NFR. By adopting the Three-Layer pattern, we can support this NFR.

In Contribution Matching method, if one NFR is associated with an Ontology Entry that is associated with a pattern or is a child of an Ontology Entry in the Argument Ontology that is associated with a pattern, we say the pattern is matched (the Argument Ontology, defined as part of SEURAT, contains a

hierarchy of arguments at different levels of abstraction). For Example, an NFR is described as “The main components of the software must be reusable” and associated with Ontology Entry “Reusability”. The Three-Layer pattern is associated with Ontology Entry “Adaptability Criteria” and affects it positively. Because in the Argument Ontology, “Adaptability Criteria” and “Reusability” have a parent-child relationship, we say the Three-Layer pattern is matched with the NFR.

When the architect selects a pattern as a candidate, the pattern is added to the rationale as an alternative solution to the decision problem. Then rationale, in the form of arguments for and against the alternative solution (pattern) is created from the pattern library. This automatic generation of Arguments will be achieved in two ways. First, if an associated Ontology Entry of a Pattern is matched with an NFR, an Argument of type Requirement is generated and this new Argument is associated with that NFR. Second, if an associated Ontology Entry of Pattern is not matched with any NFR, an Argument of type Claim is generated and the Ontology Entry is associated with the Claim. The direction of the Argument, whether the Argument is for or against the Alternative, will be determined by the direction in which the Ontology Entry affects the pattern (positively or negatively).

4. USING SEURAT_ARCHITECTURE

Figure 2 shows SEURAT_Architecture with an initial set of NFRs, the root decision, and the Pattern Library. The rationale, giving the NFRs and the decisions, is presented in the Rationale Explorer view, shown on the left-hand side of the figure. The Pattern Library is displayed on the right. The example used is the Aqua Lush architecture, described in [22].

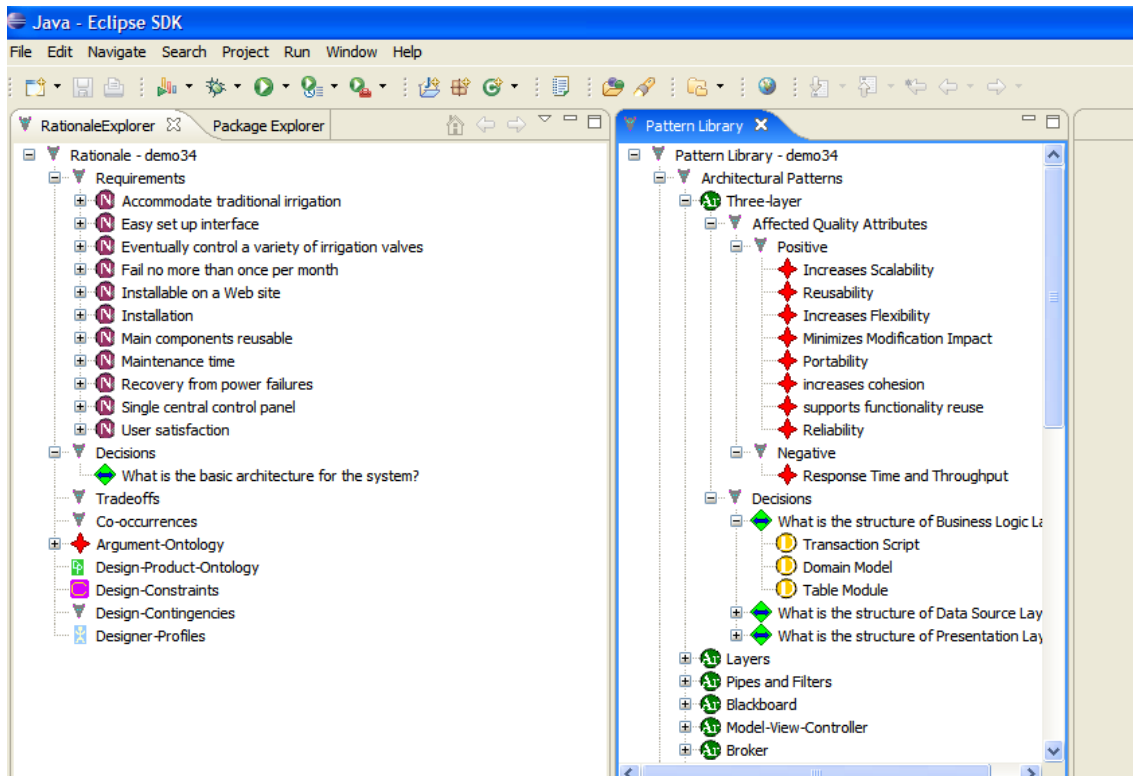


Figure 2. SEURAT_Architecture

The software architect starts the decision-making process by first using SEURAT_Architecture to create a new software architecture project. This will create a project with a single root decision—“What is the basic architecture for the system.” The software architect will use the system to enter in the NFRs for the system.

The architect right-clicks on the root decision and selects “Generate Candidate Patterns.” This brings up the “Generate Candidate Pattern” window where the architecture can select the matching method, pattern scope, and problem category. Figure 3 shows the Generate Candidate Pattern window. After entering their criteria, SEURAT_Architecture finds the patterns with the best fit and returns a list of these patterns and their evaluation scores, as shown in Figure 4.

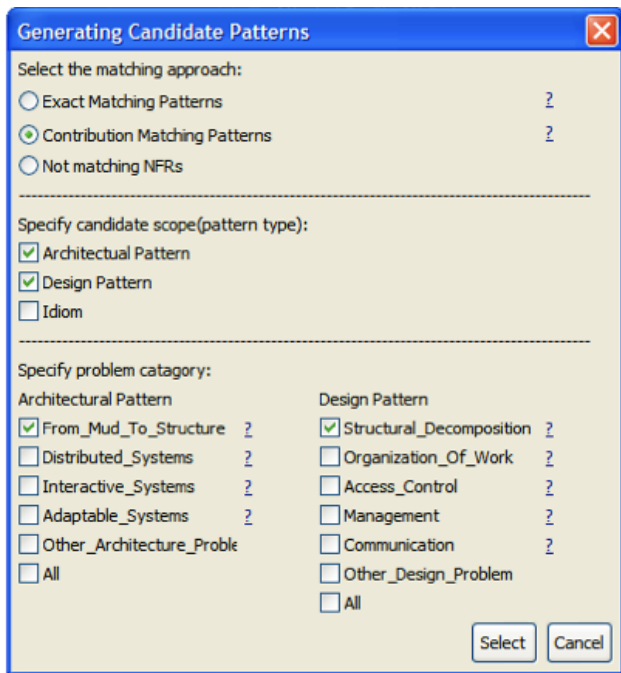


Figure 3. Generate Candidate Pattern Window

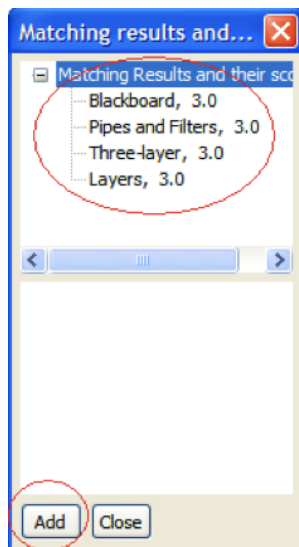


Figure 4. Pattern Alternatives

The architect can select one or more patterns from the list of candidates. These then appear in the rationale as alternatives to the selection they were in response to, in this case the root decision. The alternatives are displayed along with their rationale, which is a combination of claims coming from the Pattern Library and the NFRs that were used to select the pattern. Figure 5 shows some of the rationale generated when the four patterns shown in Figure 4 are all selected as alternatives. The icons, which are easier to read in color than the black and white, are a triangle for Alternatives, a star for Arguments (with a red or green border indicating negative or positive support for the alternative), a square with a “C” for Claims, and a circle with an “N” in it for NFRs. The darker colored stars are the Argument Ontology entries (where IS or NOT indicates if the claim does or does not support the criteria).

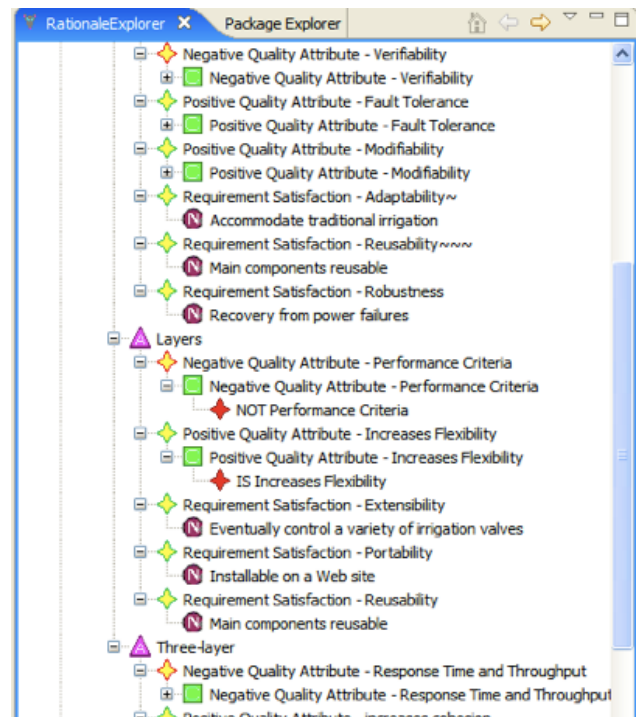


Figure 5. Rationale After Selecting Candidate Patterns

SEURAT_Architecture will use the arguments for and against each alternative (pattern) to calculate a score for each one of them using a weighted sum measurement, which takes into account the priorities (weights) of each quality attribute and NFR. This information is available to the architect to assist them in deciding which pattern alternative should be selected for the decision. After a pattern is selected, SEURAT_Architecture will generate the decisions that need to be made to continue the design based on that pattern, as defined in the Pattern Library. For example, if the Three-Layer Model is selected, the architect then needs to design the Business Logic Layer, the Data Source Layer, and the Presentation Layer. Each of these has a set of alternatives and rationale that are also imported automatically from the Pattern Library.

As the design continues, the architect can augment the rationale provided by SEURAT_Architecture with their own arguments.

The result will be a fully specified rationale for the final system architecture.

5. CONCLUSIONS AND FUTURE WORK

SEURAT_Architecture supports the software architect in selecting architectural patterns, design patterns, and idioms for use in the architecture. It also captures the rationale for each pattern using the NFRs for the architecture under development and the rationale stored in the pattern library. The initial design decision is guided using the NFRs for the specific decision. Later decisions are guided using information in the pattern library. Future versions of this software will use the "Generate Candidate Patterns" functionality for later decisions as well as the initial one.

The system is currently under evaluation with experiments being run using software architects using SEURAT_Architecture to assist in decision-making and rationale capture (the experimental group) and using software architects using SEURAT as rationale capture tool but without the pattern-based features (the control group).

Decisions made during architectural design are critical to the success of the system. Architects may not always be aware of all the architectural patterns available to them and why one might be preferred over another. Those implementing or modifying the architecture may not understand where or why a pattern is used or the reasons why it was chosen. SEURAT_Architecture assists with these two problems by providing tool support in pattern selection and by capturing the rationale for architectural decisions.

6. ACKNOWLEDGMENTS

This work is supported by NSF CAREER Award CCF-0844638 (Burge).

7. REFERENCES

- [1] Ali-Babar M.A., Gorton I., Jeffery R., 2005. Capturing and Using Software Architecture Knowledge for Architecture-Based Software Development. Proceedings of the Fifth International Conference on Quality Software, p.169-176
- [2] Ali-Babar, Wang X., Gorton I. (2005), PAKME: A Tool for Capturing and Using Architecture Design Knowledge. 9th International Multitopic Conference, IEEE INMIC 2005
- [3] Avgeriou P., Zdun U. 2005. Architectural patterns revisited: a pattern language. 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July.
- [4] Bass L., Clements P., Kazman R., 2003. Software architecture in practice, 2nd edition, Addison-Wesley, Reading, MA;
- [5] Bass, L., Clements, P., Nord, R.L., Stafford, J. 2006. Capturing and using rationale for a software architecture. In: Dutoit, A.H., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering. Springer, pp. 275-292.
- [6] Beck K., Johnson R., 1994. Patterns Generate Architectures, Proceedings of the 8th European Conference, ECOOP '94, Lecture Notes in Computer Science, Volume 821.
- [7] Boehm, B., & Kitapci, H. 2006. The WinWin approach: using a requirements negotiation tool for rationale capture and use. In Rationale Management in Software Engineering (Dutoit, A., McCall, R., Mistrik, I., & Paech B., Eds.), pp. 173-190. Heidelberg: Springer-Verlag.
- [8] Burge, J., Brown, D.C. 2004. An Integrated Approach for Software Design Checking Using Rationale, In Proc. of Design Computing and Cognition '04, J. Gero (Ed.), Kluwer Academic Publishers, Netherlands, 557-576
- [9] Burge, J., Brown, D.C., 2006. Rationale-Based support for software maintenance, In: Dutoit, A., McCall, R., Mistrik, I., Paech, B. (Eds.), Rationale Management in Software Engineering, Springer-Verlag, Berlin, pp. 273-296.
- [10] Burge, J. E. and Brown, D. C. 2008. SEURAT: integrated rationale management. In Proceedings of the 30th international Conference on Software Engineering (Leipzig, Germany, May 10 - 18, 2008). ICSE '08. ACM, New York, NY, 835-838.
- [11] Burge, J.E., Carroll, J.M., McCall, R., Mistrik, I., 2008 Rationale-Based Software Engineering, Springer.
- [12] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. 1996, Pattern-Oriented software architecture, Volume 1: A System of Patterns. Wiley.
- [13] Capilla R., Nava F., Duenas J.C., 2007. Modeling and Documenting the Evolution of Architectural Design Decision (SHARK/ADI'07), ICSE Workshops, IEEE CS
- [14] Chung, L. Representing and Using Non-Functional Requirements: A Process-Oriented Approach, Ph.D. thesis, Univ. of Toronto, 1993.
- [15] Chung, L., Nixon, B., Yu, E., 1995, Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design. Proc. 1st Int. Workshop on Architectures for Software Systems
- [16] Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., 2000. Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers.
- [17] Cui X., Sun Y., Mei H. 2008, Towards Automated Solution Synthesis and Rationale Capture in Decision-Centric Architecture Design, Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008): pp.221-230
- [18] Davis, A. 1993. Software Requirements: Objects Functions and States. Prentice Hall
- [19] Dutoit A. H., McCall R., Mistrik I. et al. 2006. Rationale Management in Software Engineering: Concepts and Techniques, in Dutoit, A.H.; R. McCall & I. Mistrik et al., Rationale Management in Software Engineering, Springer Berlin Heidelberg, pp. 1-48
- [20] Dutoit, A., McCall, R., Mistrik, I. and Paech, B. (Eds.) Rationale Management in Software Engineering, Springer-Verlang, Berlin, 2006
- [21] Falessi D., Becker M., Cantone G. 2006. Design decision rationale: experiences and steps ahead towards systematic use. ACM SIGSOFT Software Engineering Notes 31(5): (2006)

- [22] Fox, C. 2006. Introduction to Software Engineering Design: Processes, Principles, and Patterns with UML2. Addison Wesley.
- [23] Fowler M.. 2003. Patterns of Enterprise Application Architecture. Addison-Wesley, Reading, MA
- [24] Gamma E., Helm R., Johnson, R. Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- [25] Harrison, N. and Avgeriou, P. 2007. Pattern-Driven Architectural Partitioning: Balancing Functional and Non-functional Requirements. In Proceedings of the Second International Conference on Digital Telecommunications Washington, DC, 21
- [26] Horner J., Attwood, M.E. 2006. Effective design rationale: understanding the barriers. In Rationale Management in Software Engineering (Dutoit, A., McCall, R., Mistrik, I., & Paech B., Eds.), pp. 73-90. Heidelberg: Springer-Verlag.
- [27] Jan S. van der Ven, Anton G. J. Jansen J., Nijhuis A. G., Bosch J. 2006. Design Decisions: The Bridge between Rationale and Architecture, in Dutoit, A.H.; R. McCall & I. Mistrik et al., Rationale Management in Software Engineering, Springer Berlin Heidelberg, pp. 329-346
- [28] Jansen AGJ, Bosch J., 2005. Software architecture as a set of architectural design decisions. In: 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), Pittsburgh, PA
- [29] Keller, S.E.; Kahn, L.G.; Panara, R.B. 1990. Specifying Software Quality Requirements with Metrics. in Thayer, R.H.; Dorfman. M.: System and Software Requirements Engineering, IEEE Computer Society Press, Washington, pp. 145-163
- [30] Kim S., Kim. D-K., Lu L., Park S. 2009. Quality-driven Architecture Development Using Architectural Tactics, J. Syst. Softw. 82, 8 (Aug. 2009), pp. 1211-1231.
- [31] Kunz, W., Rittel, H. 1970. Issues as elements of information systems. Working Paper 131, Center for Urban and Regional Development, University of California, Berkeley
- [32] Lee J. 1991. Extending the Potts and Bruns model for recording design rationale, Proceedings of the 13th International Conference on Software Engineering (ICSE '13), IEEE Computer Society Press, Los Alamitos, CA, pp. 114-125
- [33] Lee, J. & Lai, K.-Y. 1991. What's in Design Rationale? Human-Computer Interaction special issue on design rationale 6(3-4) pp. 251-280
- [34] Lee, J. 1997. Design rationale systems: understanding the issues. *IEEE Expert: Intelligent Systems and Their Applications*, 12, 3 (May. 1997), 78-85.
- [35] MacLean, A., Young, R.M., Bellotti, V., Moran, T.P., 1996. Questions, options and criteria: elements of design space analysis, In: Moran, T., Carroll, J. (Eds.), Design Rationale Concepts, Techniques, and Use, Lawrence Erlbaum Associates, NJ, 1996, pp. 201-251.
- [36] MacLean, A., Young, R. M., & Moran, T. P. 1989. Design rationale: The argument behind the artifact. Proceedings of the CHI '89 Conference on Human Factors in Computing Systems, 247-252. New York: ACM.
- [37] McCall, R., 1987. PHIBIS: Procedural hierarchical issue-based information systems. In: Proceedings of the 13th International Congress on Planning and Design Theory, pp. 17-22.
- [38] Nilsson J. 2006. Applying Domain-Driven Design and Patterns: With Examples in C# and .NET, Addison Wesley Professional.
- [39] Pena Mora F., Vadhavkar S. 1996. Augmenting Design Patterns with Design Rationale. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11(2): pp. 93-108
- [40] Potts, C., Bruns, G., 1988. Recording the reasons for design decisions. In: Proceedings of the 10th International Conference on Software Engineering, pp. 418-427.
- [41] Tang T., Babar M.A., Gorton I., Han J. 2006. A Survey of Architecture Design rationale. *Journal of Systems and Software*, 79(12): pp. 1792-1804
- [42] Tang A., Jin Y., Han J., 2007. A rationale-based architecture model for design traceability and reasoning. *The Journal of Systems and Software* 80, pp. 918-934. Elsevier
- [43] Tang A., Han J., Vasa R. 2009. Software Architecture Design Reasoning: A Case for Improved Methodology Support, *IEEE Software*, vol. Mar/Apr 2009: pp. 43-49, 2009
- [44] Tyree J. and Akerman A 2005. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):pp. 19-27
- [45] Zhu L and Gorton I. 2007. UML Profiles for Design Decisions and Non-Functional Requirements. in Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK-ADI), Minneapolis, MN
- [46] Zdun, U., Avgeriou, P., Hentrich, C., Dustdar, S., 2008. Architecting as decision making with patterns and primitives. SHARK 2008: 11-18
- [47] Zimmermann O., Gschwind T., Küster J. M., Leymann, F., Schuster, N., 2007, Reusable Architectural Decision Models for Enterprise Application Development. (QoSA'07), Springer-Verlag LNCS 4880, 15-32