Collaborative Software Review for Capturing Design Rationale

Philip Johnson[1]
Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
(808) 956-3489
johnson@uhics.ics.hawaii.edu

1 Introduction

Mechanisms and representations for design rationale can be broadly broken down into three (overlapping) categories: a historical record of the process of design, a justification for why a design decision satisfies the behavioral requirements for the system, and a description of how a design choice relates to other potential alternatives [Lee, 1990].

An alternative classification of design rationale mechanisms can be based upon their temporal relationship to the artifacts being designed. For example, some design rationale mechanisms, such as Object Lens [Lai *et al.*, 1988] and gIBIS [Conklin and Begeman, 1988], appear to have a *waterfall* orientation toward design—in other words, they view it as an activity that precedes implementation of the artifact. Alternatively, [Potts and Bruns, 1988] extends the original gIBIS model to a more *concurrent* orientation toward design as an activity intermixed with implementation.

We have recently begun a research project concerned with support for collaborative software review as an exploratory activity. Through these activities, we have found several interesting relationships between collaborative software review and design rationale. Not surprisingly, collaborative software review activities provide an excellent source of design rationale information, since analysis of artifacts under review often involves design justifications and alternatives. Also, software reviews form an interesting point at which to take a discrete "snapshot" in the historical record of design, since they tend to take place only when the software is in a relatively consistent and coherent state. Finally, our exploratory focus leads to a interesting variation on concurrent orientations toward design and implementation.

Despite these relationships, little emphasis upon design rationale appears in review methodologies (and their more formal counterpart, inspections), in part because of their strong focus upon detection and removal of errors. In this paper, we briefly describe some of the relationships between collaborative review and design rationale, and discuss our nascent project to bring these two areas of research together in a useful fashion.

[1]Support for this research was provided in part by the National Science Foundation Research Initiation Award CCR-9110861 and the University of Hawaii Research Council Seed Money Award R-91-867-F-728-B-270.

2 The Exploratory Nature of Design Rationale

Much of the research on design rationale focusses upon development of a representation language for design artifacts, justifications, alternatives, and evaluations. These languages can be evaluated with respect to their expressiveness, human usability, machine interpretability, and so forth. [Lee, 1990] evaluates several current design rationale representations along these dimensions, and concludes that these dimensions are somewhat mutually exclusive. In other words, optimizing a design rationale representation for machine interpretability ordinarily requires some sacrifice in human usability, since machine interpretation requires disambiguation, increased representational structure, and attachment of formal semantics inconsistent with the high-level, intuitive abstractions often preferred by people. A representation optimized for machine interpretability may also lead to restrictions on the kinds of relationships expressed within the representation. Similar trade-offs occur when optimizing along the other dimensions.

One conclusion from this analysis is that there exists no single "holy grail" design representation language, but rather that the most appropriate language depends upon the context of use and associated requirements along each dimension. A natural waterfall method for choosing a design rationale language is to evaluate the application and development environment demands upon the design representation language along each dimension, and then pick an existing design rationale language that appears to make the trade-offs most appropriate for the domain.

However, in many cases it is difficult to make *a priori* decisions about the most appropriate design rationale representation. The specific kinds of representational expressiveness required from the design rationale language, the degree of user friendliness, and the requirements for machine interpretability may become clear only after the software development process is underway. Furthermore, the most appropriate trade-offs along each of these dimensions will certainly change over the course of development, as the experience of the group members (both about the application and about each other) increases. The emergent nature of the most appropriate design rationale representation motivates an incremental, exploratory approach to its development and adoption.

The next section presents one approach to accomodating some of the exploratory characteristics of design rationale. This approach is preliminary and does not resolve every important issue in design rationale; nevertheless, we believe it offers a complementary alternative to other approaches and forms a good basis for further research.

3 Exploiting Collaborative Review for Incremental Design Rationale Generation

According to [Freedman and Weinberg, 1990], a review is a way to:

1. Point out needed improvements in a product;

2. Confirm those parts of a product in which improvement is either not desired or not needed;

3. Achieve technical work of more uniform quality.

The classic work by Fagan [Fagan, 1976; Fagan, 1986] reports significant gains in the development of defect-free software on time and at lower costs. Fagan achieved these results through a very narrowly focused form of review called code inspection, involving a rigidly defined process model, roles for participants, and resulting artifacts.

In our experience of performing reviews and inspections during software development, however, we have found this limited focus on determining the presence and absence of errors unsatisfying. While determining and removing instances of errors in a single software development artifact is important, the analysis activities by participants required to perform this function necessitates much reasoning on a design rationale level that is neglected in the process and products of review. Obviously, much of this neglected information has use in future software products, or to future members attempting to "come up to speed" with the application. Such rationales can often take the form of guidelines that prescriptively direct future design and implementation activities.

A primary reason for the narrow focus of formal inspection activities is to guarantee that the intensive investment in human resources required for preparation and conduct of the review pays off in a concrete, observable, and quantifiable manner. Using traditional paper-and-pencil approaches, a less focused process may lead to increased overhead, unprofitable digressions, and low-impact activities that dilute the overall effectiveness and impact of the review. Research on the ICICLE environment [Sembugamoorthy and Brothers, 1990] has already shown that computational support for review can decrease its overhead. We believe that with additional forms of support, the inspection process could be modified in such a way as to capture additional useful information without detracting from its ability to detect and remove errors.

Based upon this conjecture, we are building a prototype collaborative review environment to support both the discovery and removal of errors as in traditional code review, and the discovery and articulation of guidelines and associated rationale information as a co-product. These guidelines and rationales are intended to be incrementally refined over the course of future reviews and development activities, and evolve through various stages of formality and machine interpretability.

To illustrate the kind of support we hope to provide, consider the following simple scenario involving the incremental generation and refinement of standards and rationale information for per-function documentation strings in a Lisp code environment.

Review 1: During an early review, a participant notes that certain functions do not have an associated documentation string. This error is recorded in our collaborative review environment in a manner similar to formal code inspection procedures. In addition, our environment will support the generation of a structured guideline representing the fact that all functions must have documentation strings, along with links to examples and counter-examples of this guideline from the reviewed system. The guideline will also includes a rationale for this decision, such as a comment to the effect that documentation strings facilitate interactive development.

Review 2: In a subsequent review, all functions happen to have associated documentation strings, so the guideline generated in Review 1 is satisfied. However, in this review a participant suggests that the guideline should be refined to specify that the names of all the parameters, their expected types, and the return values and types must be included in the documentation string. The corresponding rationale might state that the lack of explicit type declarations and the expressive nature of polymorphism in Lisp make this information unavailable through automated code analysis tools, though crucial when potential users of the code assess its applicability. The environment supports the incremental refinement of the initial guideline by adding these additional conditions on the appropriate form for documentation strings, along with examples and counterexamples of its use.

Review n: After several more rounds of review, the commenting guideline is found to be very helpful, but due to the ever-present pressures on developers, not always correctly followed. Furthermore, as the system grows in size, the cumulative overhead on the review process of checking for this issue may become prohibitive. We intend the environment to support gathering of metrics useful for assessing this issue, such as the number of times this violation has been discovered and a log of reviewer time spent recording instances. Based upon this information, a decision could be made to partially automate the process of ensuring conformance to this guideline. For example, an Emacs macro could be implemented to parse source code and check to see that all functions have documentation strings, and that each documentation string mentions all of the parameter names, the substring "return value", and legal type specifiers. After implementing such a mechanism, the documentation guideline would be further refined to indicate that successful use of this code checker is required before releasing code to future inspections.

Review m: At some later point, a new project begins that uses C++ as its implementation language. The reviewers adapt the guidelines generated from the first project to the second, using the rationale information to guide the process. For example, the first rationale for use of documentation strings applies equally well to C++ as to Lisp.[2]

[2]Well, per-function documentation remains useful, even though C++ does not provide as convenient a means to retrieve it interactively.

However, the use of type declarations in C++ implies that the rationale for the appearance of parameter names and their types in the documentation is no longer valid, and thus the corresponding guideline should be removed.

While simple, this example illustrates some of the top-level requirements for our method. First, the introduction of computer support for the inspection process means that the primary emphasis of developers should remain on the discovery and removal of errors, in keeping with the literature. In addition, mechanisms to abstract from the occurrence of a single error to a more general representation of its meaning and relevence should be provided.

Second, the environment should document guidelines and associated rationales through explicit reference to specific examples and counterexamples from the actual artifacts under review. (Other categories, such as exceptions, where the guideline is violated in an acceptable manner are also possible.) Such linkages can be made with relatively little overhead in an inspection environment, since errors are naturally discussed in the context of specific examples.

Third, the environment should support an incremental, exploratory, and emergent approach to the generation and refinement of guidelines and rationales. While some bootstrapping of the guidelines is possible, the orientation is on acquiring this information as a natural by-product of the review and inspection process, then facilitating the use of it in future development activities. The example illustrates how some guidelines can be initially specified at a high level of abstraction, then successively refined as further experience suggests improvements, and finally automated if the gains appear to warrant the effort.

Fourth, the guidelines should live beyond the scope of an individual review session or even individual development project. This is in contrast to conventional review activities, where the focus on discovery and removal of individual errors in individual documents leads to little explicit transfer of information across review instances within or between projects.

We believe that this approach to automated support of software review can support the collection of design rationale information along all three of the categories mentioned in the introduction. First, the sequence of review activities and results should form a useful record of the process of design as it co-occurs with implementation. Second, a typical use of the rationale description for a guideline is to document its relationship to the behavioral requirements for the system. Third, the use of examples, counter-examples, and exceptions provides an easily obtained, though only partial description of the design space.

The process-level aspects of our approach differs significantly from other design rationale mechanisms, which are typically employed during the construction of the artifact. In waterfall-oriented methods; the design process completely precedes the construction of the artifact, in more concurrent approaches, the design process is intermixed, but in both cases, the rationale information is captured before or during implementation of the artifact. In contrast, our approach combines a concurrent model with a process whereby rationale infor-

mation is actually captured *after* construction of the artifact. In other words, we believe that analysis of the implementation can serve as a fruitful source of design-level information.

4 Status and Future Directions

We are currently refining the requirements for the prototype implementation through the conduct of small-scale reviews without automated support. We hope to begin implementation of a prototype this summer, which will use the EGRET framework for exploratory group work [Johnson, 1992] for its low-level platform. The EGRET framework provides a flexible environment for specification and evolution of exploratory collaborative systems.

This research raises several open questions that we hope to begin answering through interactions with other researchers at this workshop:

- Though most design rationale representations focus upon explaining the motivation behind a particular design artifact, our focus is on rationale to explain guidelines that control and direct future design or implementation activities. This is a subtle but significant shift in emphasis. What are the strengths and weaknesses of current design rationale representations in this application context?
- How can traditional design rationale information arising *a priori* be integrated with the *a posteriori* rationales arising from this process? Is a uniform representation for both types required?
- How do you specify constraints in an incremental form that allows automation and internal as well as external consistency checking? Must the guidelines always be automated in an ad-hoc, procedural manner?
- As reviews progress within a project, the set of guidelines and rationales tend to become increasingly domain and group specific. How can the transition to a new project be accomplished while maintaining as much useful information as possible?

References

[Conklin and Begeman, 1988] Jeff Conklin and Michael L. Begeman. Gibis: a hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331, 1988.

[Fagan, 1976] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[Fagan, 1986] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[Freedman and Weinberg, 1990] D. Freedman and G. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews.* Dorsett House, 1990.

[Johnson, 1992] Philip Johnson. Supporting exploratory CSCW with the EGRET framework. In *Proceedings of the 1992 Conference on Computer Supported Cooperative Work*, July 1992. Also published as Technical Report CSDL-92-01, University of Hawaii Department of Information and Computer Sciences.

[Lai *et al.*, 1988] Kum-Yew Lai, Thomas W. Malone, and Keh-Chiang Yu. Object lens: a "spreadsheet" for cooperative work. *ACM Transactions on Office Information Systems*, 6(4):332–353, 1988.

[Lee, 1990] J. Lee. What's in design rationale? *Human-Computer Interaction*, 6(3-4), 1990.

[Potts and Bruns, 1988] C. Potts and G. Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering*, 1988.

[Sembugamoorthy and Brothers, 1990] V. Sembugamoorthy and L. Brothers. ICICLE: Intelligent code inspection in a C language environment. *The 14th Annual Computer Software and Applications Conference (COMSPAC)*, October 1990.