An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer

Lin Qiu and Christopher K. Riesbeck
Department of Computer Science
Northwestern University
Evanston, Illinois 60201 USA
{qiu, riesbeck}@cs.northwestern.edu

Abstract: Individualized feedback is an important factor in fostering effective learning. It is, however, often not seen in schools because providing it places a significant additional workload on teachers. One way to solve this problem is to employ critiquing systems. Critiquing systems, however, require significant development effort before they can be put into use. In this paper, we describe an incremental approach that facilitates the development of educational critiquing systems by integrating manual critiquing with critique authoring. As a result of the integration, the development of critiquing systems becomes an evolutionary process. We describe a system that we built, the Java Critiquer, as an exemplar of our model. Results from a pilot test and real-life usage of the system have shown that the system successfully provides a setting for accumulating critiques and at the same time supporting teachers in critiquing student code.

Introduction

Constructivist learning theory views learning as a process where the learner actively constructs knowledge and understanding (e.g., Bransford, Goldman, & Vye, 1991; Brown, 1988; Chi et al., 1994). Feedback has been recognized as an important element in fostering this process (Bransford, Brown, & Cocking, 1999). Individualized feedback can provide learners with alerts to problematic situations, relevant information to the task at hand, directions for improvement, and prompts for reflection. Such feedback is rarely seen in traditional school settings where teachers actively deliver the knowledge to students through lectures and students passively listen. Students' major tasks are listening, taking notes and answering questions. Though they have opportunities to ask questions, continual one-to-one attention from teachers is usually not available. Contrarily, in an apprenticeship setting (Collins, Brown, & Newman, 1989), where students engage in a problem-solving task and their work is critiqued regularly by teachers, individualized feedback is often seen and plays an important role in the process. Feedback provides students with scaffolding in performing the tasks and makes sure they cover all the necessary learning materials. A major obstacle in adopting the apprenticeship approach in school is that reviewing student work and personalizing critiques is labor-intensive and time consuming.

One way to avoid the above problems is to build critiquing systems to help teachers critique student work. Critiquing systems have been developed in various domains, e.g., kitchen design (Fischer et al., 1993), trauma care (Gertner, 1995), multimedia authoring (Nakakoji et al., 1995), and proved effective in providing users with useful contexualized information such as flaws in design, comparison to previous cases, and possible alternatives (Silverman, 1992). Extending this line of work into educational settings makes it possible to dramatically improve individualized feedback and attention in school.

In this paper, we describe an incremental approach that facilitates the development of educational critiquing systems by integrating manual critiquing with critique authoring. We start by outlining the challenges in developing critiquing systems and how our model addresses these issues. Next, we describe the interface and architecture of the critiquing system that we built, called Java Critiquer, as an exemplar of our model. We present results from a pilot test and real-life usage of the system at the end.

Problem

Two common approaches to critiquing are comparative critiquing and analytic critiquing (Robbins, 1998). Comparative critiquing compares user's work with a presumably better solution in the system and points out differences between them. Systems that use stored solutions supplied by experts have a limited problem space. Users can only work on tasks for which solutions have already been generated. Systems that can automatically generate a solution based on their extensive knowledge about the domain are more powerful but only feasible in limited well-understood domains. Focusing on differences from exemplary solutions can often lead users to imitate the already stored solutions, inhibiting innovative problem solving. Since generating one solution already involves a complicated endeavor, producing multiple solutions becomes incredibly difficult.

Analytic critiquing uses rules to detect critiquing opportunities. It does not require reasoning, planning, goal representations or complete knowledge of the domain to generate a solution in order to produce critiques. Critiques are generated when certain rules are activated. Compared to comparative critiquing, analytic critiquing can be applied to a broader range of domains. Incremental development of the system can also be made possible when critiquing rules are implemented independently.

Education covers a broad range of domains. Problems in many domains do not have one single right answer. With our goal to provide practical and inexpensive systems to facilitate educational critiquing, rule-based critiquing better fits our needs. There are still, however, a number of challenges in the traditional way of developing critiquing systems (see Figure 1):

- In order to support purely computer-based critiquing, the vocabulary of operations and situations in the system has to be specified in advance so that rules can be written to provide accurate and timely feedback. Once deployed, there is no easy way to adjust the existing content or incorporate new information into the system. Such closed systems are not ready to be widely used for educational purposes since the focus and scope of critiquing may need to be changed from time to time between different contexts.
- To build fully autonomous systems that perform accurate and comprehensive critiquing, developers have to work with domain experts to make sure the systems have complete coverage of all possible mistakes and their corresponding critiques. This requires significant upfront design, implementation, and piloting of the systems before they can be put into use. Such requirement inevitably increases the difficulty of building these systems.
- It is difficult for experts to anticipate all the mistakes that novices commonly make. Asking experts to supply all possible mistakes will result in unnecessary effort spent on cases that rarely happen. More importantly, commonly occurring critical cases may fail to be collected.

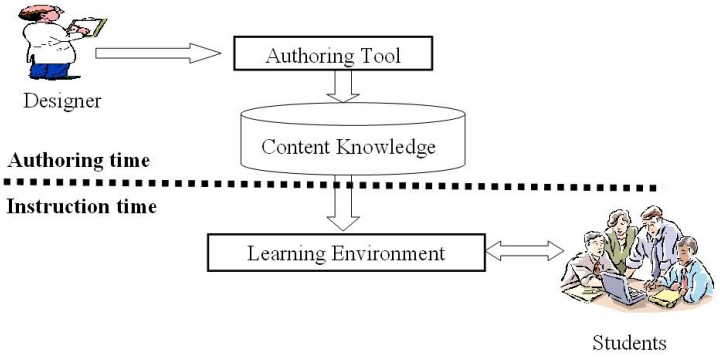

Figure 1: The traditional development model of critiquing systems.

Approach

Based on our observations of critiquing in a real world educational setting, we believe critiques develop through several stages. First, a teacher sees a mistake in a student's solution and writes a specific critique. After critiquing the same mistake repeatedly in different forms and contexts, the teacher improves his or her understanding of the nature of the mistake and how to respond to it. The teacher sometimes forms general patterns for quickly recognizing the mistake in different contexts. With practice, the teacher optimizes this pattern and can

very quickly recognize and critique the mistake. Finally, a critique becomes reliable enough so that the teacher can publicize it for use by other teachers, assistants, or by learners for self-assessment. These stages are show in Figure 2. Not all stages occur for all critiques, and different critiques will be at different points in the lifecycle at any given time.

Stage	Activity
Realize	Identify a specific instance of a mistake and critique it.
Familiarize	Repeatedly critique the same mistake in different forms and contexts repeatedly
Generalize	Gradually realize the nature of the mistake and generate a general and pedagogically sound critique or a set of related critiques.
Optimize	Learn patterns for recognizing the mistake in various forms and contexts, and critique it promptly.
Publicize	Distribute the pattern and critique to students for self-assessment or share them with public.

Figure 2: The natural critiquing development process.

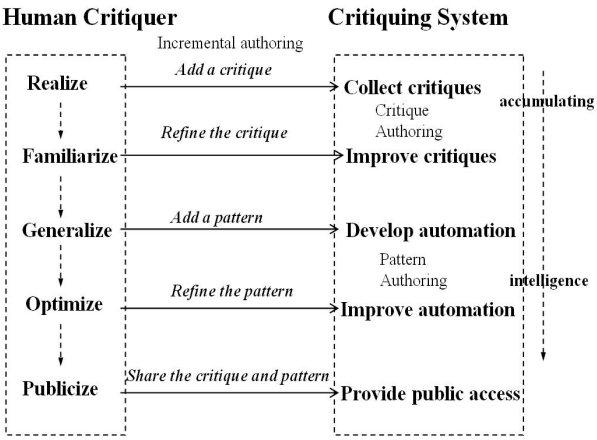

Figure 3: An incremental development model of critiquing systems.

Based on the above observation, we propose a model (see Fig. 3) that supports the natural critiquing development process by allowing the teacher to incrementally author critiques in a critiquing system during on-going use. In our approach, a teacher is part of a critiquing system's feedback loop. Using a database of common critiques, the system uses pattern matching to automatically critique a student's solution. The teacher reviews the critiques, modifying or removing inappropriate ones as needed. Using the same database, the teacher manually inserts additional critiques for mistakes not recognized by the current patterns, creates critiques for mistakes not previously seen, and optionally adds patterns to existing critiques that failed to match or matched incorrectly. The key point is that *authoring* is integrated with *usage*, so that usage guides improvement in the accuracy and scope of automatic critiquing.

In this way, the development of a critiquing system becomes an incremental process in which situations for critiquing and corresponding critiques are realized, implemented into the system, assessed through practical use, and refined based on experience. There is no need to anticipate and implement all possible critiquing situations up-front. Issues not anticipated during system design can be explored during real use. Furthermore, because a teacher is part of the critiquing loop, the system can be put into use at a much earlier stage and still deliver useful high-quality responses. Initially, the teacher does most of the work. The teacher is motivated to author critiques and critique patterns as a way to significantly reduce his or her future workload. Critique authoring is done at use time based on real examples. Instead of being built as intelligent at design time, the system gradually migrates into an intelligent system through real use.

In the following, we describe the Java Critiquer, a system that we have built and been using, as an exemplar of our model.

The Java Critiquer

A common misconception of novice computer programmers is that all that matters is that the program runs and generates the right results. This belief often leads to inefficient, unreadable, and unmaintainable code, and bad habits in software development.

A web-based critiquing system called the Java Critiquer was developed to teach students how to write clean, maintainable and efficient code. The Java Critiquer helps the teacher to detect and critique bad programming choices often seen in introductory programming courses. For example, Figure 4 shows a critique generated by the Java Critiquer on a piece of Java code.

Java code	Critique

If (length > 10)
  return true;
else
  return false;
	There is more code than you need to write. You already have a boolean value. Just write *return length > 10;* instead. You never need to write an IF to return true in one case and false in the other.

Figure 4: A piece of problematic Java code with a corresponding critique.

Interface Overview

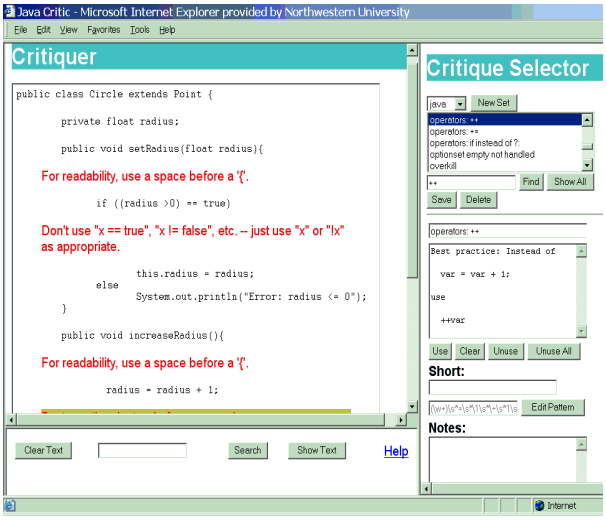

Figure 5: The interface of the Java Critiquer.

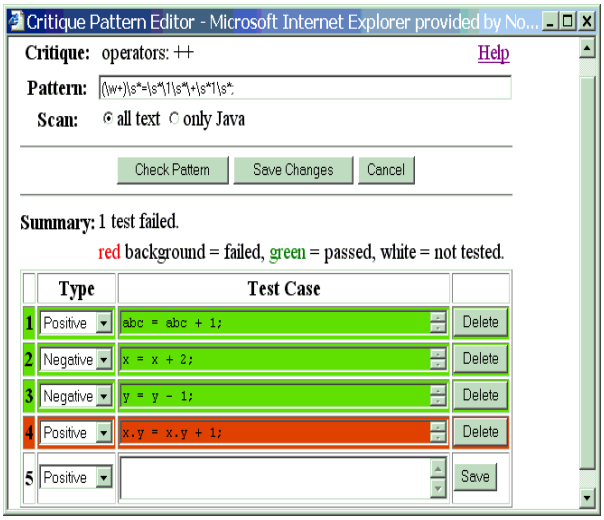

Figure 6: The interface of the Pattern Editor.

Figure 5 illustrates the interface of the Java Critiquer. There are two panels on the interface, the Critiquer panel and the Critique Selector panel. To start critiquing, the teacher pastes Java source code into the large text box in the Critiquer panel. The system performs automatic critiquing on the code using pattern matching (details will be discussed later in the paper). Critiques generated are inserted right below the problematic code lines. The teacher can click on a critique and edit the critique text, or remove it entirely. Since a large number of typical mistakes made by novice programmers appear frequently in code, handling them by automatic critiquing can significantly reduce the teacher's workload. It also reduces the chance of missing critiques. Even when automatically generated critiques require editing to make the critique more appropriate, it is still easier than searching for the mistake and writing the critique manually. Allowing the teacher to review and modify automatically generated critiques also ensures the quality of system critiquing.

After reviewing the automatically generated critiques, if any, the teacher manually critiques the code. The teacher can click on the desired line in the source code and type a critique in the text box in the Critique Selector panel. A search function is provided for the teacher to select a line by typing in a phrase. The teacher can also save a critique into the system for future use. Besides saving the effort of typing the critique again, this provides a means of building a collection of useful critiques that can be shared with other instructors, reviewed for common student difficulties, and shared with students for self-assessment.

Existing critiques can be selected by title from the critique list in Critique Selector. The teacher can also type a short phase in the find box to quickly find critiques containing that phase. The inexpensive search and select functions support critiques that are hard to implement automatically. For example, to automate critiques such as "Use more descriptive variable names," and "You repeat the same lookup over and over. Do it once and save in a variable," the system would need to have natural language understanding and code semantic analysis ability. The select and search function avoids such high-level complexity, but still facilitates teachers in applying these critiques.

The teacher can automate a critique by attaching a pattern to it. There are two types of patterns in the system for matching Java. One type of pattern is written in regular expression. Regular expressions are a way of writing string patterns. They are powerful but can get quite complex. It is common to discover a pattern either fails to match in places where it should, or matches in many places where it should not. A pattern editor (Fig. 6) is provided for editing and testing patterns. The teacher can specify test cases for the pattern. A test case is a short example of text for testing the pattern. A positive test case is one where the pattern should match. A negative test case is one where the pattern should not match. Test cases of both types are helpful for debugging patterns, recording particularly tricky cases, or even pointing out cases that cannot be handled. The system automatically matches the pattern against the test cases and highlights each test case with either red or green to indicate whether or not the test result complies with the teacher's expectation. This approach of using test cases to work out the correct pattern resembles the machine learning model that utilizes positive and negative cases to converge a pattern over time. The Pattern Editor saves the teacher from switching to another application for testing. It also serves as a documentation facility allowing other teachers to review all the test cases for a pattern, therefore helping them to understand the capability of the pattern.

While regular expression patterns are concise, they are fairly cryptic and clumsy. For example, the regular expression pattern for matching code in the form of "var = var + 1;" is "(\w+)\s*=\s*\1\s*\+\s*1\s*;" (The critique attached to the pattern is "Best practice: Instead of var = var + 1; use ++var.") Furthermore, regular expression is unable to create patterns for matching arbitrary nested constructs such as parenthesized arithmetic expression. Another type of pattern, the JavaML (Badros, 2000) pattern is used in the system to overcome such limitation. A prototype of the JavaML pattern editor has been developed. It allows the teacher to create patterns by tying in Java source code with variables embedded. For example, the teacher can type in "$x = $x + 1;" and indicate "$x" is a pattern variable. The system then automatically generates the JavaML pattern that matches code in the same form, e.g., "a = a + 1." Compared to authoring regular expression patterns, authoring JavaML patterns is simpler and much more direct.

The use of the Java Critiquer presents a practical approach because each stage in system development made by the teacher is motivated by its immediate benefit. The teacher adds a critique into the system to save the effort of typing a common response repeatedly. This leads to a database of reusable critiques. The teacher creates a pattern for a critique when finding and applying the critique repeatedly becomes tedious and time-consuming. The teacher refines a pattern when false matches are frequent enough to require significant additional effort to effect a remedy. This in itself motivates the teacher to gradually improve the intelligence in the system in order to reduce his or her workload.

System Architecture

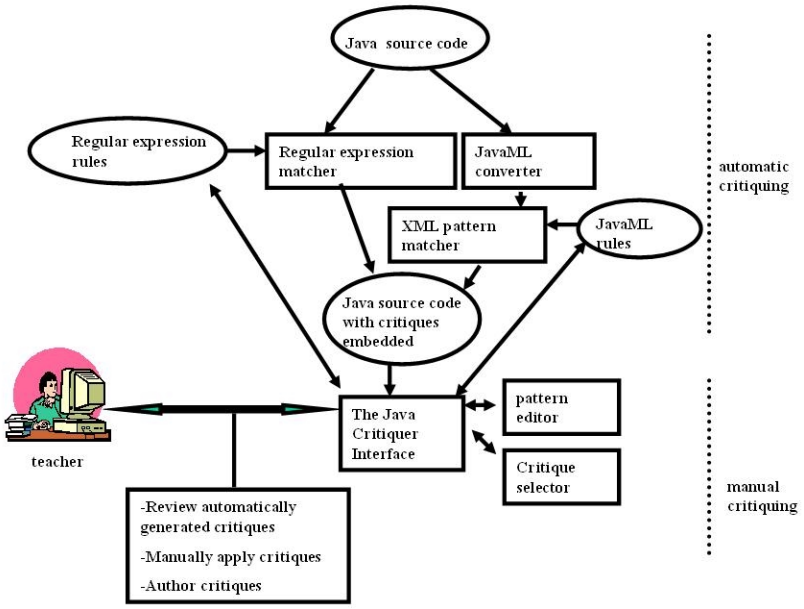

Figure 7: The system architecture of the Java Critiquer.

The system architecture of the Java Critiquer is illustrated in Figure 7. The usual web-based client-server model was employed to provide easy access to the system with little overhead. The system reads in Java source code from the teacher and starts two rule-based matching processes. One process converts the source code into JavaML and uses the XML pattern matcher to match XML patterns against the JavaML code. The other process uses the regular expression pattern matcher to match regular expression patterns against the source code. Critiques attached with patterns matched in either process are inserted into the source code. Interface tools such as Critique Selector and Pattern Editor facilitate the teacher applying or authoring the critiques in the system.

The system uses a framework that can be customized to support critiquing in other domains. A Lisp parser has been added into the system with Lisp patterns for critiquing Lisp code. It is not describe here in detail as it is not the focus of this paper. A more detailed description of its implementation can be found in Qiu and Riesbeck (2003). Related results will be shown in the Evaluation section.

Evaluation

A pilot study was conducted to assess the quality of critiques generated by the system using JavaML pattern match. There were a total 19 critiques implemented using JavaML pattern. The Java Critiquer was used on 28 programs selected randomly from student homework submissions in two introductory Java courses. Our domain expert, a computer science professor, ranked each computer-generated critique with "Good" or "Inappropriate." "Inappropriate" indicates critiques that are either wrong, or not worth saying in the given context.

Among the total of 205 critiques that were generated, 97% were "Good" and 3% were "Inappropriate" (Fig. 8). On average, each program still received 7 valid critiques. With such performance, we believe the use of the system can be helpful in providing feedback to students as well as saving teachers work.

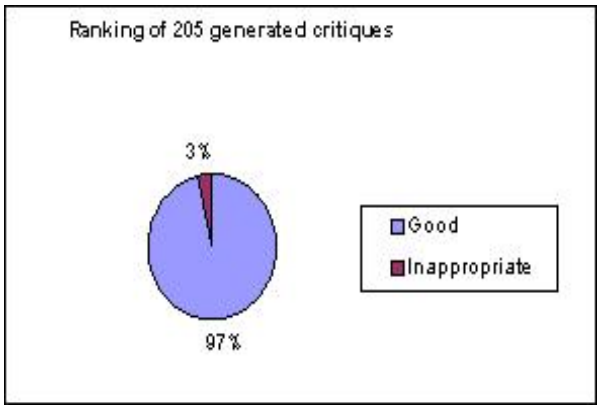

Figure 8: The ranking of 205 generated critiques using JavaML patterns.

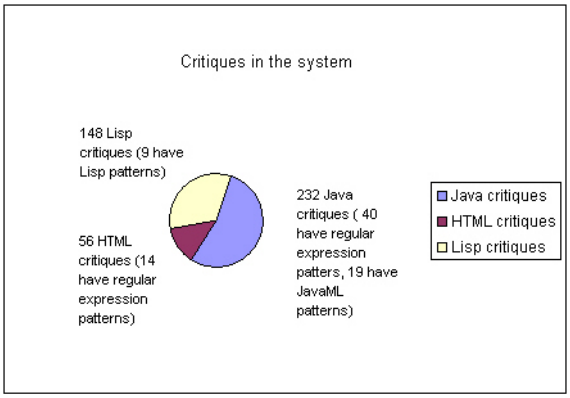

Figure 9: Critiques in the system.

The Java Critiquer has been used by two teachers, together, for over a year in critiquing Java, HTML and JSP code in university-level introductory programming courses. A total of 436 critiques have been collected in the system (Fig. 9). Of these, 236 critiques are for Java. Forty Java critiques have regular expression patterns. Nineteen Java critiques have JavaML patterns. There are 56 critiques for HTML with 14 of them having regular expression patterns. There is also a Lisp Critiquer. It has 148 critiques for Lisp. Nine of them have Lisp patterns. The constant usage of the system suggests our model is practical and beneficial for supporting teachers in providing individualized feedback to students.

Note that only a small number of Lisp critiques has patterns. This is because most Lisp critiques with patterns have already been thoroughly tested and offered to students for self-assessment. We plan to release the Java Critiquer to students in the future as well. The student version will only have automatic critiquing with highly reliable critiques.

Related Work and Discussion

Just as 'design for testing' affects system design, so does 'design for incremental authoring.' First, the system architecture has to provide a place for humans in the loop. That means that the system needs to be able to display inputs and responses to authors in a readable form. It needs to allow humans to easily modify those responses, as needed, before returning them to the end user. It also needs to allow the authors to modify the processes that generate those responses. That in turn implies data-oriented process controls. In additional, we assume systems designed for incremental authoring do not use complex domain models to generate feedback. This can reduce the up-front development effort and make it easy to modify and augment computer feedback generation.

Seeding, evolutionary growth, reseeding (SER) is a model describing three stages in the evolutionary development of software systems (Fischer & Ostwald, 2002). Seeding is the first stage where a system is created with initial knowledge that enables the system to be used for practice. Evolutionary growth is where the system supports user work and collects information generated by its use. Reseeding is where information collected during evolutionary growth is formalized and organized to support the next cycle of development. While our model also uses an evolutionary approach, it does not have a separate stage of reseeding. Critiquing rules collected by the system are already reusable. Individual rules can be refined independently at use time. There is no need for an explicit optimization stage.

A number of rule-based critiquing systems have been developed to support learning by doing, e.g., Lisp-Critic (Fischer, 1987). All of them, however, assume that a substantial set of critiques is developed at design time. Our model can function without complete knowledge for critiquing. The knowledge acquisition process takes place gradually with use.

Intelligent tutoring systems (Wenger, 1987) employ detailed student models in providing individualized feedback to the student. They ask a student to solve a specific problem, and analyze the solution to update and refine an internal model of the student's knowledge and misconceptions. Tutoring rules use the student model to guide the selection of feedback and future problems to pose. In order to do this, the system needs detailed knowledge of the problems, objectives of the lessons, and an overall lesson plan. In contrast, our approach is much less knowledge-intensive and problem-specific, and avoids the complexity of user modeling. We put a teacher in the loop not only to handle the difficulties, but because our experience has been that teachers want to be part of the feedback process.

In programming, there are some very useful tools, e.g., LINT (Johnson, 1978) and CodeAdvisor (Hewlett-Packard Company, 1998), that analyze code for common errors in memory management, class design and implementation, and coding style. These tools are for professional programmers, however, and give feedback inappropriate for novices just learning to program.

Conclusions

We have described a development model that allows teachers to incrementally author a critiquing system during use. Systems in our model facilitate incremental development by (1) starting with a task model-based framework (2) allowing early deployment and testing through instructor involvement (3) providing an architecture and interface for in-use authoring. We describe the Java Critiquer, a critiquing system that we built, as an exemplar of our model. The Java Critiquer helps teachers to detect and critique bad programming choices in student Java code. It allows teachers to gradually enter and update critiquing knowledge during real use of the system. Results from real-life usage have shown that the system successfully provides a setting for accumulating critiques and at the same time supports teachers in critiquing student code. We believe our model presents a practical and beneficial approach to developing critiquing systems for education.

References

Badros, G. (2000). JavaML: A Markup Language for Java Source Code. In *Ninth International World Wide Web Conference, May 2000.*

Bransford, J. D., Brown, A. L., & Cocking, R.R. (Eds.) (1999). How people learn: Brain, Mind, Experience, and School. Washington, DC. National Academy Press.

Bransford, J. D., Goldman, S. R., & Vye, N. J. (1991). Making a difference in people's ability to think: Reflections

on a decade of work and some hopes for the future. In R. J. Sternberg and L. Okagaki (Eds.), *Influences on Children* (pp. 147-180). Hillsdale, NJ: Erlbaum.

Brown, A. L. (1988). Motivation to learn and understand: On taking charge of one's own learning. *Cognition and Instruction*, 5, 311-321.

Chi, M. T. H., de Leeuw, N., Chiu, M., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, 18, 439-477.

Collins, A., Brown, J.S., & Newman, S. (1989). Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics, In L.B. Resnick (Ed.) Knowing*, Learning, and Instruction: Essays in Honor of Robert Glaser*, Lawrence Erlbaum Associates, Hillsdale, NJ.

Fischer, G. (1987). A Critic for LISP. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Milan, Italy.

Fischer, G., (1998) Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments. *International Journal of Automated Software Engineering*, Kluwer Academic Publishers, Dordrecht, Netherlands, Vol. 5, No.4, October 1998, pp. 447-464,

Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., & Sumner, T. (1993). Embedding Critics in Design Environments. *The Knowledge Engineering Review*, Vol. 8:4. 1993.

Fischer, G., Ostwald, J. (2002). Seeding, Evolutionary Growth, and Reseeding: Enriching Participatory Design with Informed Participation. In *Proceedings of the Participatory Design Conference* (PDC'02), T. Binder, J. Gregory, I. Wagner (Eds.), Malmö University, Sweden, June 2002, CPSR, P.O. Box 717, Palo Alto, CA 94302, pp 135-143.

Gertner, A. (1995) *Critiquing: Effective Decision Support in Time-Critical Domains.* Ph.D. Dissertation, Dept. of Computer and Info. Science., Univ. Of Pennsylvania.

Hewlett-Packard Company. (1998). SoftBench SDK: CodeAdvisor and Static Programmer's Guide. HP Part Number: B6454-90005, URL: http://docs.hp.com/hpux/onlinedocs/B6454-90005/B6454-90005.html

Johnson, S.C. (1978). Lint, a C Program Checker. Unix Programmer's Manual. AT&T Bell Laboratories: Murray Hill, NJ.

Nakakoji, K., Reeves, B. N., Aoki, A., Suzuki, H., & Mizushima, K. (1995). eMMaC: Knowledge-Based Color Critiquing Support for Novice Multimedia Authors. *Proceedings of ACM Multimedia '95*, San Francisco.

Qiu, L., and Riesbeck, C. K. (2003*).* Facilitating Critiquing in Education: The Design and Implementation of the Java Critiquer. In *Proceedings of the International Conference on Computers in Education* (ICCE), Hong Kong, December 2003.

Robbins, A. E. (1998). *Design Critiquing Systems*. Tech Report UCI-98-41. Available at http://www.ics.uci.edu/~jrobbins/papers/CritiquingSurvey.pdf

Sefika, M., Sane, A., & Campbell, R. H. (1996). Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany.

Silverman, B. (1992). Survey of Expert Critiquing Systems: Practical and Theoretical Frontiers. *CACM*, Vol.35, No.4.

Wenger, E. (1987) *Artificial Intelligence and Tutoring Sustems: Computational and Cognitive Approaches to the Communication of Knowledge*. Los Altos, CA: Morgan Kaufmann Publishers, Inc.