Making Critiquing Practical: Incremental Development of Educational Critiquing Systems

Lin Qiu
Department of Computer Science
Northwestern University
Evanston, Illinois 60201 USA
847-467-1619
qiu@cs.northwestern.edu

Christopher K. Riesbeck
Department of Computer Science
Northwestern University
Evanston, Illinois 60201 USA
847-491-7279
riesbeck@cs.northwestern.edu

ABSTRACT

Expert critiquing systems in education can support teachers in providing high quality individualized feedback to students. These 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 real-life usage of the system suggest benefits for supporting teachers in critiquing student code.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education. D.2.2 [**Software Engineering**]: Design Tools and Techniques - *evolutionary prototyping*. H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *prototyping*. I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems.

General Terms

Design, Human Factors.

Keywords

Intelligent assistants, critiquing, incremental authoring, evolution, educational software, education.

1. INTRODUCTION

Intelligent user interfaces for critiquing systems for performance support have proved effective in providing users with useful contexualized information such as design flaws and possible alternatives [8]. Meanwhile, in education, individualized feedback has been recognized as an important factor in fostering effective learning [2, 3]. It can provide learners with alerts to problematic situations, relevant information to the task at hand, directions for improvement, and prompts for reflection. It is not, however, often seen in schools because reviewing student work and personalizing critiques is labor-intensive and time consuming.

The need for providing teachers with practical assistance in critiquing has led to our work in developing critiquing systems for education. In this paper, we describe an incremental approach that facilitates the development of educational critiquing systems by integrating manual critiquing with critique authoring.

2. 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 1. Not all stages occur for all critiques, and different critiques will be at different points in the lifecycle at any given time.

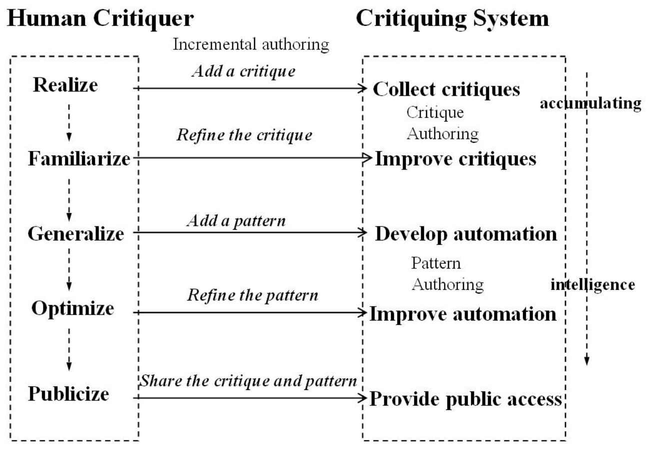


Figure 1. Incremental development of critiquing systems.

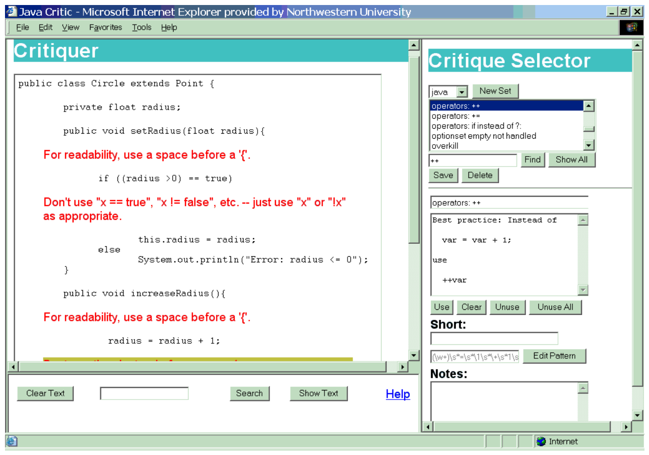
Based on the above observation, we propose a model that supports the natural critiquing development process by allowing the teacher to incrementally author critiques in a critiquing system during on-going use. In our approach, a teacher is part of a critiquing system's feedback loop. Using a database of common critiques, the system uses pattern matching to automatically critique a student 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 in 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 the teacher's 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.

3. THE JAVA CRITIQUER

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.



```
public class Circle extends Point {

        private float radius;

        public void setRadius(float radius){

    For readability, use a space before a '{'.

                if ((radius >0) == true)

    Don't use "x == true", "x != false", etc. -- just use "x" or "!x"
    as appropriate.

                        this.radius = radius;
                else
                        System.out.println("Error: radius <= 0");
        }

        public void increaseRadius(){

    For readability, use a space before a '{'.

                radius = radius + 1;
```

operators: ++
operators: +=
operators: if instead of ?:
optionset empty not handled
overkill

++

operators: ++

```
Best practice: Instead of

  var = var + 1;

use

  ++var
```

(\w+)\s*=\s*\1\s*\+\s*1\s

Figure 2. The interface of the Java Critiquer.

Figure 2 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 student Java code into the large text box in the Critiquer panel. The system performs automatic critiquing on the code using pattern matching. 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, this is still easier than searching for the mistake and writing the critique manually. Allowing the teacher to review and modify automatically generated critiques ensures the quality of system critiquing.

After reviewing the critiques generated by the system, the teacher performs manual critiquing on the code. The teacher can insert a critique by selecting an existing critique from the database, or typing in a new one. Searching and editing tools in the Critique Selector panel help the teacher find, use, and edit existing critiques and patterns. New critiques can be just added to the database. Optionally, the teacher can attach a pattern to enable automatic critiquing. Currently, two types of patterns are supported, general regular expressions, and JavaML [1] patterns. Regular expression patterns are applied directly to the Java source, and useful for short text segments. JavaML patterns are applied to the output of an internal Java parser, and useful for matching larger Java structures. To support the incremental authoring of patterns based on experience, the built-in pattern editor lets the teacher attach to each pattern examples of code that it should and should not match. The system automatically highlights each test case with either red or green to indicate whether the test result complies with expectation. The integration of test cases supports optimizing the patterns. It also helps publicize the database by documenting through examples the intent of both the pattern and the associated critique. Further publicizing occurs when the more reliable pattern-based critiques are made accessible through a web interface to students for self-assessment. A student interface lets students run the automatic critiques on their code themselves, reducing turn-around time and teacher effort even more.

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 over and over again. 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 effort for remedy. Thus, the teacher is motivated to gradually improve the intelligence in the system in order to reduce workload.

4. EVALUATION

The Java Critiquer has been used by two teachers together for over a year for critiquing Java, HTML and JSP code in university-level introductory programming courses. A total of 436 critiques have been collected in the system (Fig. 3). 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. 9 of them have Lisp patterns. The constant usage of the system suggests our model is practical and beneficial for supporting teachers to provide individualized feedback to students.

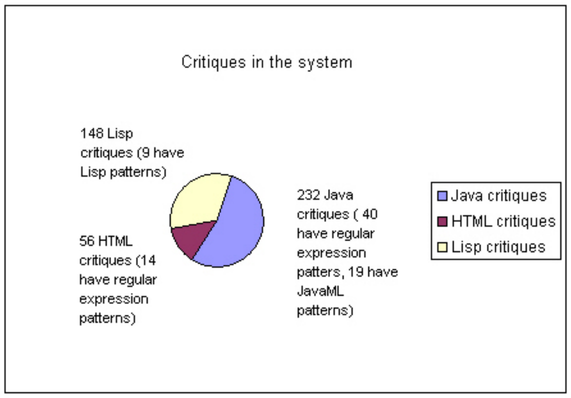


Figure 3. Critiques in the Java Critiquer.

5. RELATED WORK AND DISCUSSION

A number of rule-based critiquing systems have been developed to support learning by doing, e.g., Lisp-Critic [4]. All of them however assume 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 [9] employ detailed student models to provide individualized feedback to the student. They asked 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 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 hard parts, 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 [7] and CodeAdvisor [6], 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.

Seeding, evolutionary growth, reseeding (SER) is a model describing three stages in the evolutionary development of software systems [5]. 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 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.

6. CONCLUSION

We have described a development model that allows teachers to incrementally author a critiquing system during use. 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. We believe our model presents a practical and beneficial approach to developing critiquing systems for education.

7. ACKNOWLEDGMENTS

We would like to thank Paul Kleczka, Joshua Ochs and Zhi Qiu who helped to implement the current design.

8. REFERENCES

[1] Badros, G. (2000). JavaML: A Markup Language for Java Source Code. In *Proceedings of the Ninth International Conference on the World Wide Web*, Amsterdam, The Netherlands, May 13-15, 2000.

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

[3] 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.

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

[5] 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,

[6] Hewlett-Packard Company. (1998). *SoftBench SDK: CodeAdvisor and Static Programmer's Guide.* HP Part Number: B6454-90005.

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

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

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