Jl. of Interactive Learning Research (2008) **19***(1)*, 119-145

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

LIN QIU
State University of New York - Oswego, USA
lqiu@oswego.edu

CHRISTOPHER RIESBECK
Northwestern University, USA
c-riesbeck@northwestern.edu

Individualized feedback is an important factor in fostering learning. However, it is often not seen in schools because providing it places considerable additional workload on teachers. One way to solve this problem is to employ critiquing systems. These systems, however, require significant development effort before they can be put into use. In this article, we describe an incremental authoring model 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. The model explores a vision for developing educational critiquing systems with much less upfront development effort. We describe a system that we built, the Java Critiquer, as an exemplar of our model. Evaluation results from performance testing and real-life usage of the system have shown that the system successfully provides a tool for accumulating critiques and at the same time helping teachers critique student code.

Introduction

A significant amount of research has shown that learning is a process where the learner actively constructs understanding rather than passively receives knowledge (e.g., Bransford, Goldman, & Vye, 1991; Brown, 1988; Chi, de Leeuw, Chiu, & LaVancher, 1994). Feedback has been recognized as an important element in fostering this process (Bransford, Brown, & Cocking, 1999). Collins, Brown, and Newman (1989) proposed to have students learn in an apprenticeship setting where students constantly receive individualized feedback on their work. Such feedback includes alerts to problematic situations, relevant information to the task, directions for improvement, and prompts for reflection. It helps students perform their learning tasks and make sure they reach expected learning goals.

The apprenticeship teaching approach is rarely seen in traditional school settings. This is because reviewing student work and personalizing feedback is labor-intensive and time consuming. Repeatedly addressing the same mistakes can also be tedious. With a large number of students in class, teachers cannot afford the amount of time and effort required in providing one-to-one attention to each student.

One way to avoid the problem is to use critiquing systems to provide feedback. Critiquing systems are a type of software that analyzes the work of its users and provides suggestions for improvement. They can report design flaws, point out incompleteness, suggest alternatives, or offer heuristic advice. Critiquing systems have been developed in domains such as medical therapy planning and computer aided design. Extending this line of work into educational settings makes it possible to dramatically improve individualized feedback and attention in school.

While critiquing systems have been proved effective in providing feedback, authoring remains a big problem. Many critiquing systems assume the critiquing knowledge in the systems are fully implemented before the systems are put into use. This involves significant upfront development effort and difficulty. It makes it hard for critiquing systems to be easily developed and widely used by teachers.

To address the above problems, this article describes an incremental model for authoring educational critiquing systems. It integrates manual critiquing with critique authoring so that the upfront development effort can be amortized over use time. The model was applied in developing a critiquing system in the computer programming domain called the Java Critiquer. In the following, we start by an introduction of critiquing systems, followed by a discussion of challenges in the development of critiquing systems. Then, we describe the incremental authoring model and explain how the model addresses the challenges. Next, we describe the interface of the Java Critiquer and its system architecture to illustrate how the model can be implemented. We present some preliminary evaluation results of the system from performance testing and real-life usage. At the end, we discuss related work in the area of software development methodologies and educational critiquing systems.

The Critiquing Approach

Critiquing systems were first developed in medical domains. They help physicians evaluate their treatment plans by providing information regarding potential problems and improvements. A critiquing system takes a problem and a proposed solution as inputs, and produces critiques of the proposed solution. This approach is different from the approach used in traditional expert systems where the system takes a description of a problem and generates a solution. The difference is important because medical problems can have multiple solutions. Physicians have subjective preferences and do not like to follow the solutions given by the system (Teach & Shortliffe, 1981). The traditional expert system approach gives user the impression that the system "tells" the user the right thing to do, whereas the critiquing approach provides non-confrontational support by assuming the user is capable to make design decisions. It lets the user determine whether the advice given is applicable. In addition to the medical domains, critiquing systems have been developed in other domains such as software engineering, desktop publishing, and kitchen design (Robbins, 1998).

Comparative critiquing and analytic critiquing are two common approaches used in critiquing (Robbins, 1998). Comparative critiquing compares users' work with presumably better solutions in the system and points out the differences. The comparison can focus on the differences in the results produced by two solutions or the differences in the solutions themselves. For example, CodeLab (Turingscraft, 2005) is a system that compares the output of users' programming code to the standard output in the system and provides feedback on errors in the code. In contrast, ONCOCIN (Langlotz & Shortliffe, 1983) is a system that compares a doctor's plans for treatment of cancer patients with the plan generated by the system. Its critiques explains the system's solution in each difference. Similarly, TraumaTIQ (Gertner, 1995) is system that critiques plans for treatment of medical trauma cases. It analyses the doctor's plan to infer its goals and generates a plan for the same goal. It then compares the generated plan with the doctor's plan and uses concise natural language critiques to indicate critical differences.

ONCOCIN and TraumaTIQ use system generated solutions for comparison. Other systems have used predefined solutions for comparison. For example, Archie (Pearce, Goel, Kolodner, Zimring, Sentosa, & Billington, 1992) is a system that helps architects evaluate building designs. It uses case-based critiquing which retrieves lessons of past relevant cases and uses them as critiques to point out potential problems in the current design.

Different from comparative critiquing, analytic critiquing uses rules to generate critiques. When certain features in user' work trigger critiquing rules, critiques associated with the rules are generated. For example, JANUS (Fischer & McCall, 1989) is a critiquing system embedded in a kitchen design environment. It has rules about building codes, safety standards, and functional preferences. When users' design violates a constraint or design principle defined in a rule, the system displays a critique explaining the problem. eMMaC (Nakakoji, Reeves, Aoki, Suzuki, & Mizushima, 1995) is another system that uses rules about color perception and color theory to critique problems in users' multimedia authoring regarding the use of color combinations and balance.

Analytic critiquing does not need complex domain knowledge to generate a solution. This allows it to be applied in domains where knowledge is not completely available or hard to implement. Compared to analytical critiquing, comparative critiquing using system generated solutions is only feasible in well-understood domains. It requires extensive domain knowledge and established problem-solving strategies including reasoning, planning, and goal representations to generate solutions. Comparative critiquing using predefined solutions can only handle problems for which solutions are already available. It requires experts to supply exemplary solutions for all of its problems.

While analytical critiquing is applicable in a broad range of domains, it is not easy to author. Experts need to write rules for all the problems in all situations. In contrast, comparative critiquing allows authors to write down problems and answers and the system takes care of comparison and feedback generation. The authoring is relatively intuitive and straightforward for experts than generating rules.

The analytical critiquing approach is chosen for our critiquing system described below despite its authoring disadvantages. This is because our critiquing system is in the computer programming domain. Complete knowledge for automatic generation of solutions is not available. Although case-based critiquing can be used to provide predefined cases as critiques, helping authors create indexing rules and similarity metrics for case retrieval however remains a problem. Using analytical critiquing allows us to start with relatively low-effort rule authoring and we believe it and can overcome the disadvantages well enough to be practical and useful in the short term.

There are two critiquing strategies used in critiquing systems, active and passive (Fischer, Nakakoji, Ostwald, Stahl, & Sumner, 1993). Active critiquing continuously monitors user activities and notifies the user as soon as a critique is generated. Passive critiquing requires the user to explicitly invoke the critiquing process. Critiquing does not occur until the user chooses to. Passive critiquing is than active critiquing less intrusive because it lets the user to control when to critique. It is, however, often found to be not activated early enough to prevent users from continuing with potential problems.

Our critiquing system is for educational purposes. It uses passive critiquing because there is no need to prevent students from making mistakes. In fact, effective learning occurs when students make mistakes and correct them by themselves (Schank, Fano, Bell, & Jona, 1993). Students learn and realize the utility of knowledge through such a process. Passive critiquing provides such an opportunity for learning and allows students to concentrate on their programming tasks without intrusion.

In the following, we discuss the difficulties in the development of critiquing systems.

Development Difficulties

Many critiquing systems developed so far are built using a model that we call the *upfront development model* (see Figure 1). In such a model, developers work with domain experts to find out all the common problems in the domain and implement all the critiques addressing these problems at design stages. Authoring tools help developers or experts enter critiquing knowledge into the systems. Then, the systems are put into use. They are expected to provide fully automatic critiquing and handle all situations by themselves. They are built as static in the sense that the knowledge and critiques in the systems are not expected to change after deployment. There are a number of challenges in using this upfront development model.

First, to build fully autonomous systems that perform accurate and comprehensive critiquing, developers have 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 complexity and difficulty of development.

Second, it is difficult for experts to anticipate all the mistakes that students commonly make. Experts might be able to identify mistakes in student work, but it is much harder for them to recall or predict all the mistakes. Significant expertise in the problem domain as well as experience working with students are required to know all the mistakes and appropriate critiques for them. Furthermore, totally depending on experts to anticipate all possible mistakes may cause unnecessary effort to be spend on cases that rarely happen or commonly occurring critical cases fail to be collected.

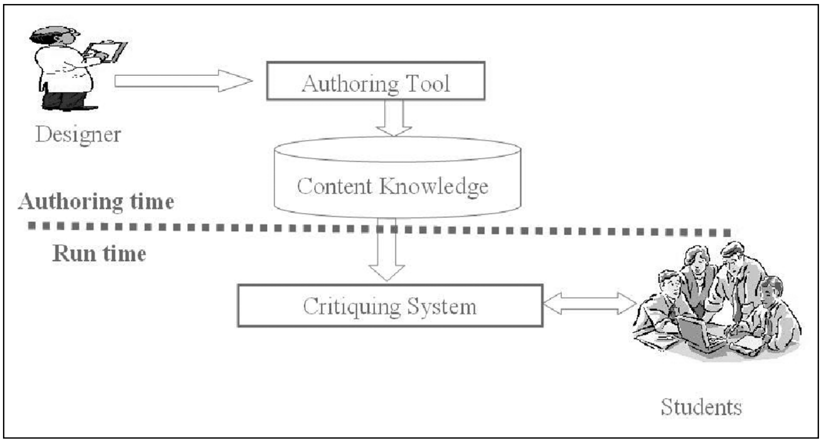


Figure 1. The upfront development model of critiquing systems

Third, there is a disadvantage of using computers to provide critiquing. It was reported by Reeves and Nass (1996) that computer programs can easily lose credibility if inappropriate feedback is noticed by users. When users have low trust of the computer program, they pay little attention to even the critiques that are appropriate. Thus, it is extremely important to produce feedback with high accuracy. This significantly increases the difficulty of building systems with coaching capability. For example, intelligent tutoring systems, which provide learners individualized coaching, are estimated to require two hundred hours of development for one hour of instruction (Woolf & Cunningham, 1987).

Fourth, 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. Once deployed, there is no easy way to adjust existing content or incorporate new knowledge into the system. This makes it difficult for critiquing systems to be adapted by teachers for use in different contexts. Teachers need to be able to change the scope or focus of the critiquing based on student performance or learning goals.

Finally, in the upfront development model, the benefit of using the system can only be obtained at a much later stage after the system is put into use. The risk of investing considerable effort with benefits remaining uncertain make instructors hesitant to put in their time and effort. This hinders the development and use of critiquing systems in a large scale.

To avoid the above difficulties, we developed an incremental authoring model that includes a teacher in the feedback loop to complement automatic critiquing and perform critique authoring. We describe this model in the following.

Incremental Authoring Model

In our observations, critiquing develops through several stages in real-world educational settings. First, a teacher sees a mistake in a student's solution and writes a specific critique. After repeatedly critiquing the same mistake in different forms and contexts, the teacher improves his or her understanding of the nature of the mistake and learns how to generate a better response to the mistake. Gradually, the teacher forms a general pattern for quickly recognizing the mistake in different forms. With practice, the teacher optimizes the pattern and becomes able to quickly recognize and critique the mistake using the pattern. Finally, a critique becomes reliable enough so that the teacher shares it with other teachers, or gives it to students for self-assessment. Figure 2 shows the above lifecycle of critiquing development. Not all stages occur for all critiques, and different critiques will be at different points in the lifecycle at any given time.

To support this critiquing development process, we developed a model that allows a teacher to incrementally author critiques in a critiquing system during on-going use (see Figure 3). In our model, the system starts with an

Stage	Activity
Realize	Identify a specific instance of a mistake and critique it.
Familiarize	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 critiquing it promptly.
Publicize	Distribute the pattern and critique to students for self-assessment or share them with public.

Figure 2. The critiquing development process

environment for critiquing but no critiques in it. When the teacher finds a mistake in students' work, the teacher critiques the mistake and can save the critique into the system for future reuse. For example, if the teacher sees the problematic Java code "if (found == true) return length;" the teacher can write "the form 'if (found == true) return length;' misses the power of boolean values. Just write 'if (found) return length;'" and save a critique for future use. By making it easy for the teacher to save during critiquing, the system gradually collects a database of critiques. Later, when the teacher sees a similar mistake, the teacher can retrieve a previously saved critique, modify it if necessary, and apply it to the new mistake. For example, if the teacher later sees "if (isBook == true) count++" the teacher can retrieve the critique that we mentioned above, generalize it to "a form such as 'test == true' misses the power of boolean values. Just use 'test'," and apply it to the new mistake. By allowing the teacher to use and refine existing critiques in the system, the system gradually improves the quality of its critiques. When a mistake appears frequently, the teacher can create a pattern for the system to automatically critique the mistake using pattern matching. For example, the teacher can create a pattern "== true" for the mistake "test == true" shown above. Using patterns, the teacher adds automatic critiquing to the system. The teacher can review critiques automatically applied by the system by modifying or removing inappropriate ones, and manually inserting additional ones for mistakes not recognized by the system. More importantly, the teacher can correct the patterns that result in false or missing matches. For example, when the teacher finds the pattern "== true" doesn't match the code "y==true" because there's no space after the '==,' the teacher can change the pattern to "==\s*true" to match any code with zero or more spaces after "==." By allowing teachers to test and correct patterns in the system, the system improves the accuracy of its automatic critiquing. When critiques and patterns are considered accurate after being tested on a fair amount of student work, the teacher can make them public by either publishing them as guidelines for common problems, or delivering them in a system where students can use for self critiquing. Figure 4 shows the steps in the critiquing system development model corresponding to the steps in human critiquing develop-

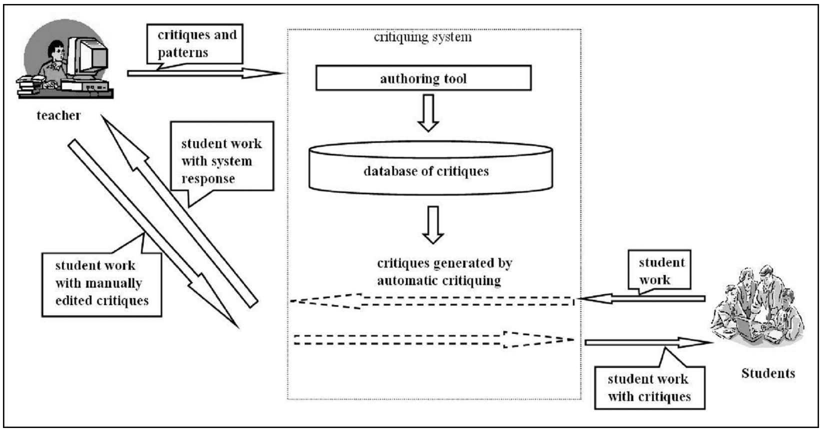

critiques and patterns
teacher
critiquing system
authoring tool
database of critiques
critiques generated by automatic critiquing
student work with system response
student work with manually edited critiques
student work
student work with critiques
Students

Figure 3. The incremental development model of critiquing systems

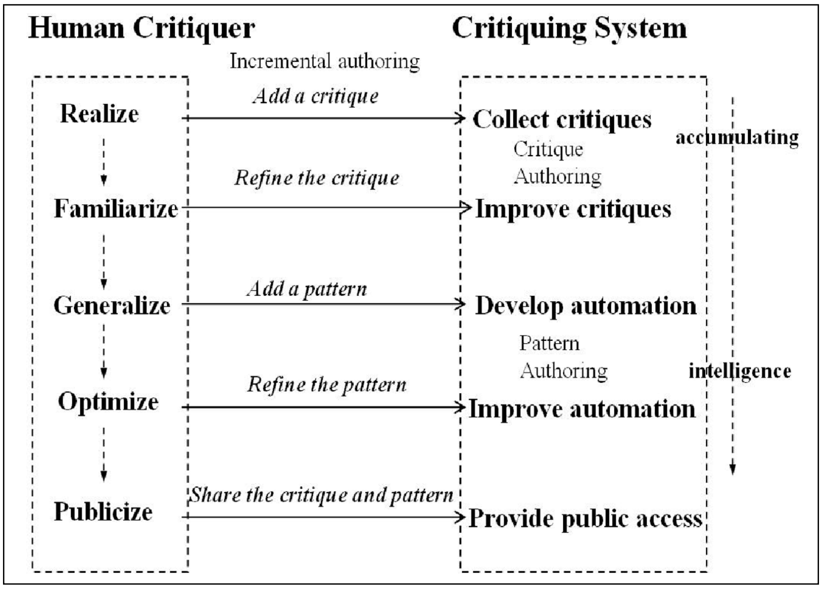


Figure 4. The incremental development model of critiquing systems supporting the natural critiquing development process

ment.

In our model, 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. The key point is that *authoring* is integrated with *usage*, so that usage guides improvement in the accuracy and scope of automatic critiquing. The critiquing system plays two roles simultaneously. On one hand, it helps the teacher critique student work. One the other hand, it serves as a vehicle for collecting critiques. The instructor in this model also plays two roles. The instructor is a user who employs the critiquing system to help critiquing. The instructor is also a developer who improves the system according to real needs.

The incremental development model avoids the upfront development effort of critiquing systems by amortizing it into use time. Authoring is done in the context of using the system to critique student work. There is no need to implement all possible critiquing situations up-front. Issues not anticipated at design stages can be explored during real use. The system is kept from totally depending on predefined knowledge. Furthermore, because a teacher is in the feedback loop to ensure the quality of critiquing, the system can be put into use at a much earlier stage. Instead of being built as intelligent at design time, the system migrates into an intelligent system through real use.

The incremental authoring model presents a motivated development approach that is driven by real needs.

- The teacher adds a critique into the system to save the effort of typing a common critique over and over again.
- The teacher creates patterns for critiques to reduce the need to find and apply very common critiques.
- The teacher refines patterns on critiques that frequently false match and have to be deleted.

During the critiquing process, therefore, the teacher is motivated to gradually improve the intelligence in the system in order to reduce workload. Furthermore, the teacher only needs to work on those parts of the system that are working poorly. There is no wasted effort building or fixing critiques or patterns that are little used or functioning adequately. Finally, critiques and patterns are authored with concrete examples available to guide the teacher.

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

The Java Critiquer

Novice programmers often consider the only important thing about a program is having the program run and generate the right results. They ignore the importance of writing clean, maintainable and efficient code. This misconception often leads to inefficient and unreadable code as well as bad habits in software development. One cause of this misconception is that many programming exercises simply require programmers write working code but do not emphasize readability and maintainability. In addition, even when good programming principles are demonstrated, students do not necessarily know when and how to apply them.

One way to solve the above problems is to critique student code and ask students to correct their code according to the critiques. This way, students learn the use of good principles in the context of writing their own code. A similar practice of code critiquing has been seen in a process called *code review* where code is first generated and then examined by programmers or tools using a set of coding standards or requirements to ensure the quality of the code (Freedman & Weinberg, 1982). A critiquing system called the Java Critiquer was developed to help teachers detect and critique bad programming choices often seen in introductory programming courses. It teaches students how to write code that is cognitively efficient (i.e., more readable and concise) and machine efficient (i.e., smaller and faster) (Fischer, 1987). For example, Figure 5 shows some critiques generated by the Java Critiquer on student code.

Interface Overview

The Java Critiquer has a web-based user interface (see Figure 6). It has two panels, the *Critiquer* panel on the left and the *Critique Selector* panel on the right. To start critiquing, a teacher pastes student Java source code into

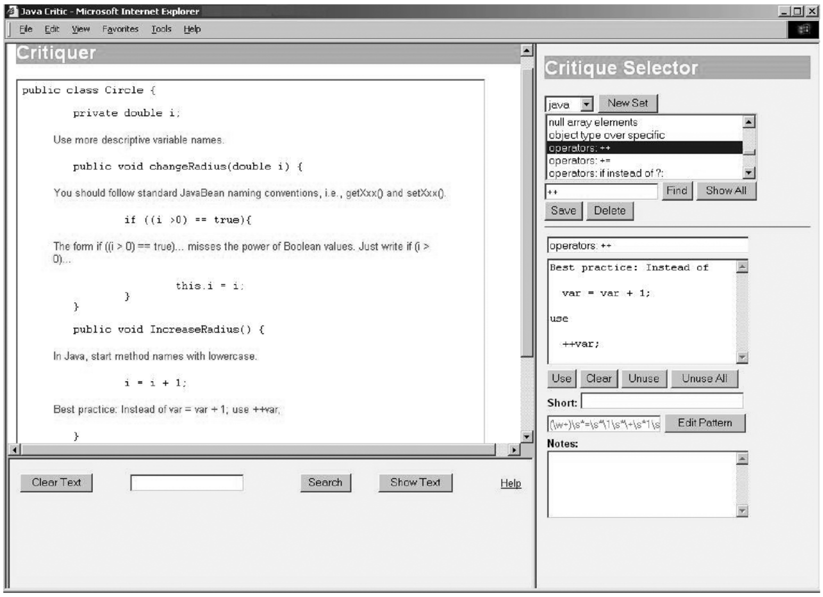
Java code	Critique

if ((x > 10) == true)
return length;
	The form *if ((x > 10) == true)…* misses the power of Boolean values.
	Just write *if (x > 10)...*

if (x > 0) {
  return true;
}else{
  return false;
}
	You never need to write an *if* to return true in one case and false in the other. Just write return *x >0*; instead.

x = x + 1;	Best practice: Instead of *var = var + 1*; use ++*var*;
return (x + y);	Those parentheses are not necessary. Just write *return x + y*;
public int x;	No public member variables! That's a cardinal rule of Java coding. Use public member functions instead.
private float x;	Use double, not float. Double represents numbers more accurately, and is the default floating point type in Java. Floats are used only rarely.

Figure 5. A piece of problematic Java code with a corresponding critique



```
public class Circle {

        private double i;

   Use more descriptive variable names.

        public void changeRadius(double i) {

   You should follow standard JavaBean naming conventions, i.e., getXxx() and setXxx().

                if ((i >0) == true){

   The form if ((i > 0) == true)... misses the power of Boolean values. Just write if (i >
   0)...

                        this.i = i;
                }
        }

        public void IncreaseRadius() {

   In Java, start method names with lowercase.

                i = i + 1;

   Best practice: Instead of var = var + 1; use ++var;

        }
```

```
null array elements
object type over specific
operators: ++
operators: +=
operators: if instead of ?:
```

Find: ++

```
operators: ++

Best practice: Instead of

  var = var + 1;

use

  ++var;
```

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

Figure 6. The interface of the Java Critiquer

the large text box in the Critiquer panel. The system performs automatic critiquing using pattern matching (details will be discussed later) and inserts critiques associated with matched patterns right below the problematic code. The teacher then verifies the generated critiques to ensure the quality of automatic critiquing. The teacher can click on a critique and edit the critique text, or remove it entirely. Since common mistakes appear frequently in novice's code, handling these mistakes by automatic critiquing can significantly reduce the teacher's workload. It also helps the teacher reduce the chance of missing mistakes. When automatically generated critiques require editing, it is still easier than reading the code to search for the mistakes and writing the critiques from scratch.

After reviewing the automatically generated critiques, the teacher performs manual critiquing. The teacher can select a line of code by clicking on the line or use the search function provided in the *Critiquer* panel by entering a phrase. The teacher can insert a critique by typing in a new one or selecting an existing one in the system. Searching and editing tools in the *Critique Selector* panel help the teacher find, use, and edit existing critiques. The teacher can select a critique by name from a list or type in a phase that the critique contains in the *find* box. These inexpensive search and select utilities help the teacher apply critiques that are hard to automate. For example, the system would need natural language semantic analysis in order to automate critiques such as "Use more descriptive variable names." The system would need data flow analysis in order to correctly apply critiques such as "You repeat the same lookup over and over. Do it once and save in a variable." Having critiques that are difficult to automate applied manually helps the system avoid functions with high complexity.

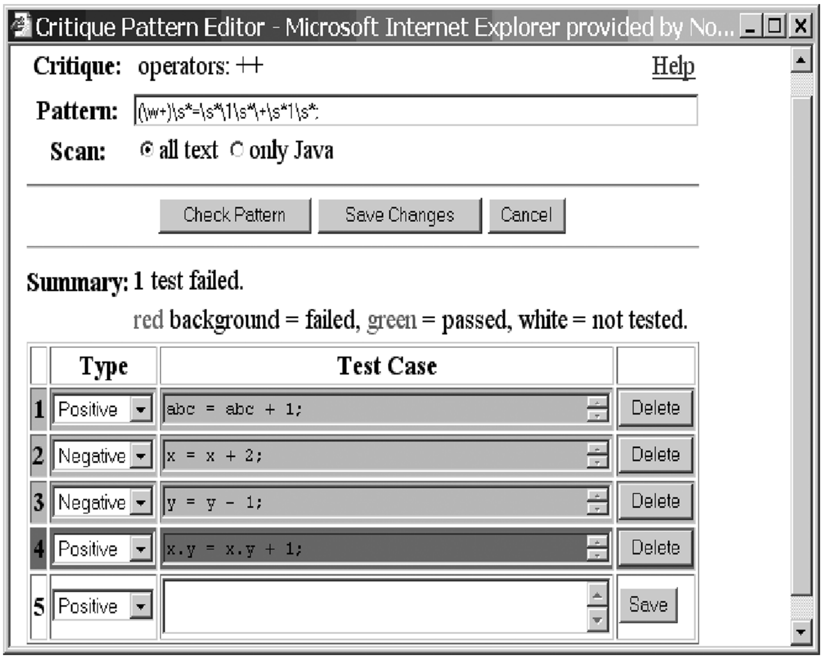
Newly created critiques can be added into the system for reuse. This saves the effort of typing them again. It also helps to build a collection of critiques that can be shared with other teachers or reviewed for common mistakes. A critique can be incorporated into automatic critiquing by creating a pattern for it. Currently, two types of patterns are supported, general regular expressions, and JavaML (Badros, 2000) patterns.

Regular expression patterns are applied directly to the Java source, and useful for short text segments. They provide a powerful way of writing patterns, and such patterns can be applied even to fragments of code. But regular expressions can quickly get quite complex. For example, in C++ and Java if a student writes "x = x + 1;" one common critique is "Instead of x = x + 1 just write ++x." The regular expression pattern for matching code in the form of *var* = *var* + *1*; is "(\w+)\s*=\s*\1\s*\+\s*1\s*;" This is not very teacher-friendly.

It is common to discover a regular expression pattern either fails to match in places where it should, or matches many places where it should not. To support the incremental authoring of patterns based on experience, a built-in pattern editor lets the teacher attach to each pattern examples of code that the pattern should and should not match (see Figure 7). Both positive and negative test cases are helpful for debugging patterns, recording particularly tricky cases, and 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 the test result complies with expectation. This approach of using test cases to work out the right pattern resembles the machine learning approach that utilizes positive and negative cases to converge a pattern over time. The integration of test cases supports the optimization patterns. It also helps to publicize the critique database by documenting through examples the intent of both the pattern and the associated critique.

Unfortunately, there's another problem with regular expressions. Regular expressions are unable to match arbitrary nested constructs such as parenthesized arithmetic expression. For example, if a student writes "return (x + 1);" a possible critique is "Those parentheses are unnecessary. Just write return x + 1;" However with normal regular expressions, you can't write a pattern that matches "return (*anything*);" where *anything* is supposed to match any expression, including code with parentheses.

The Java Critiquer uses JavaML patterns to overcome this problem. JavaML patterns are applied to the output of an internal Java parser, and useful for matching larger Java structures. (Details about JavaML patterns will


Critique Pattern Editor - Microsoft Internet Explorer provided by No...
Help
Check Pattern
Save Changes
Cancel

Critique: operators: ++

Pattern: `(\w+)\s*=\s*\1\s*\+\s*1\s*;`

Scan: [✓ all text] only Java

Summary: 1 test failed.

red background = failed, green = passed, white = not tested.

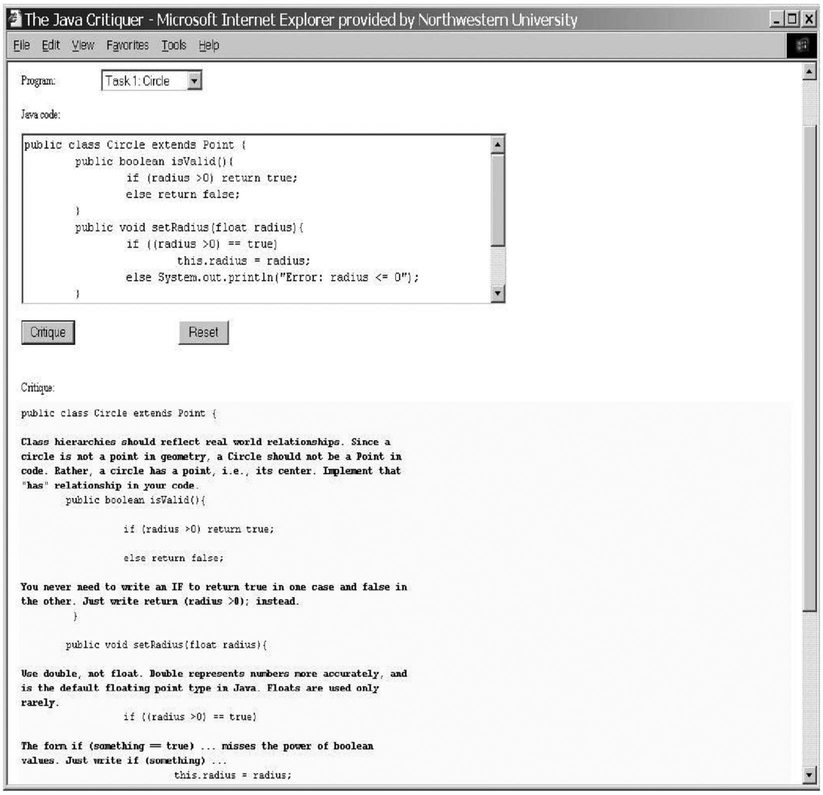
	Type	Test Case	
1	Positive	`abc = abc + 1;`	Delete
2	Negative	`x = x + 2;`	Delete
3	Negative	`y = y - 1;`	Delete
4	Positive	`x.y = x.y + 1;`	Delete
5	Positive		Save

Figure 7. The interface of the Pattern Editor

be described later). A prototype of the JavaML pattern editor allows the teacher to create patterns by writing Java source code with variables embeded. For example, the teacher can write *$x = $x + 1* and indicate that *$x* is a pattern variable. The system automatically generates the JavaML pattern for matching code such as $a = a + 1$. Compared to regular expression, the authoring of JavaML patterns is can be more direct and simpler.

When certain pattern-based critiques become reliable enough, the teacher can make them accessible through a web interface to students for self-assessment. A student interface (see Figure 8) lets students themselves run the automatic critiquing on their code, reducing turn-around time and teacher effort even more.

The Java Critiquer uses the web-based client-server architecture. It provides a web interface for teachers to use the system. Once the teacher pastes student Java source code into the web interface, the system starts two rule-based matching processes. One process converts the source code into JavaML and matches it against JavaML patterns using the pattern matcher. The other process matches regular expression patterns against the Java source code using a regular expression pattern matcher. Critiques with patterns matched in either process are inserted into the source code. Figure 9


The Java Critiquer - Microsoft Internet Explorer provided by Northwestern University
File Edit View Favorites Tools Help
Program:
Task 1: Circle
Java code:
Critique
Reset
Critique:

```
public class Circle extends Point {
        public boolean isValid(){
                if (radius >0) return true;
                else return false;
        }
        public void setRadius(float radius){
                if ((radius >0) == true)
                        this.radius = radius;
                else System.out.println("Error: radius <= 0");
        }
```

```
public class Circle extends Point {

Class hierarchies should reflect real world relationships. Since a
circle is not a point in geometry, a Circle should not be a Point in
code. Rather, a circle has a point, i.e., its center. Implement that
"has" relationship in your code.
        public boolean isValid(){

                if (radius >0) return true;

                else return false;

You never need to write an IF to return true in one case and false in
the other. Just write return (radius >0); instead.
         }

        public void setRadius(float radius){

Use double, not float. Double represents numbers more accurately, and
is the default floating point type in Java. Floats are used only
rarely.
                if ((radius >0) == true)

The form if (something == true) ... misses the power of boolean
values. Just write if (something) ...
                        this.radius = radius;
```

Figure 8. The student interface of the Java Critiquer

shows the software components in the Java Critiquer.

SYSTEM ARCHITECTURE

The Java Critiquer's architecture can be customized for supporting critiquing in other domains. A Lisp parser has been plugged into the system for using Lisp patterns to critique Lisp code. Results of its use are discussed in the evaluation section.

Critiquing Rules

Critiquing rules in the Java Critiquer are written in a specific type of XML format called LMX (Language for Mapping XML) (Maruyama, Tamura, & Uramoto, 1999). The left-hand side of a rule is a LMX pattern. In the Java Critiquer, the pattern is a JavaML pattern for matching JavaML code generated from the Java parser. The right-hand side of a rule is a critique.

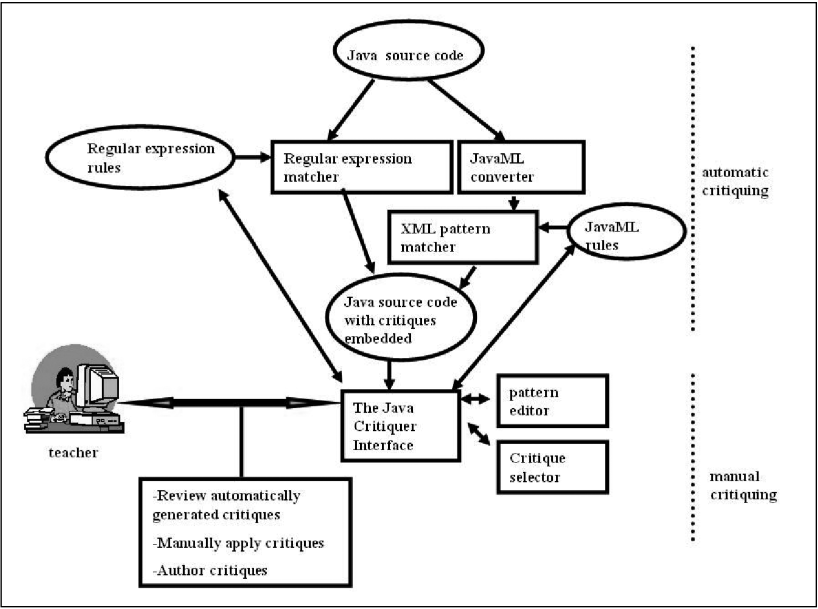


Figure 9. The system architecture of the Java Critiquer

Figure 10 shows an example of a critiquing rule. Symbols starting with a *$* sign in the pattern are variables to match parts of the JavaML, such single JavaML elements or attribute values of JavaML elements. For example, in Figure 10, *$srcBegin*; and *$srcEnd*; are variables to match attributes in JavaML referring to the beginning and ending character position of the test condition in an *if* statement. Variables and their matched value are saved in a table during the match. After the match, variables in the critique are replaced with their matched values. The use of variables allows a critique to quote student code in its text. For example, Figure 11 shows the critique generated by the rule shown in Figure 10. It includes a part of the user's code, "length > 10," in its text. This way, a general critique is customized to one that is specific to a student's code.

Pattern Matcher

In the Java Critiquer, the pattern matcher matches JavaML patterns against the JavaML tree generated from the Java parser. It starts from the root of the tree and traverses the tree matching every pattern against every node in the tree. When a pattern is matched, its corresponding critique is inserted into the Java source code.

The pattern matcher is a general matcher that can take any XML source

```
<lmx:pattern>
  <lmx:lhs>
   <if srcEnd="$srcEnd1;">
     <test srcBegin="$srcBegin;" srcEnd="$srcEnd;">
       <lmx:extension class="lmx.extension.SegmentMatch"/>
     </test>
      <true-case>
                               <return><literal-boolean value="true"/></return>
      </true-case>
      <false-case>
                               <return><literal-boolean value="false"/></return>
       </false-case>
    </if>
  </lmx:lhs>
  <lmx:rhs>
   <critique pos="$srcEnd1;">
      <text>
There is more code than you need to write. You already have a boolean value. Just write <code>return
<srcCode srcBegin="$srcBegin;" srcEnd="$srcEnd;"/>; </code>instead. You never need to write an IF
to return true in one case and false in the other.
     </text>
    </critique>
   </lmx:rhs>
</lmx:pattern>
```

Figure 10. A critiquing rule

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* statement; to return true in one case and false in the other.

Figure 11. A piece of problematic Java code with a corresponding critique

as input. It has been used experimentally in the math domain to critique students' mathematical proofs written in MathML. A proof of concept critiquer for UML models was also developed using the same matcher. The domain independent feature of the matcher makes it reusable in other domains.

Specialized Java functions can be called in the matcher to check for conditions including logical connectives ("and", "or" and "not") and ones that are difficult to express in XML. For instance, a Java function is built to detect if the first character of a string is in lower case. It is used in a pattern to detect class or interface names starting with a lower-case character, which violates standard Java naming conventions. These Java functions are loaded during runtime using the Java reflection API when the pattern matcher detects "extension" tags. These extension functions are saved in individual Java files rather than hard-coded into the matcher. This extension mechanism is similar to the software plug-in architecture where components can be added without changing the host program. It allows rule writers to create their own Java functions and use them in their rules without changing the matcher.

Evaluation

Two evaluation studies were conducted with the Java Critiquer. One was on the performance of JavaML critiquing rules. The other was on the usability for incremental authoring.

In the performance evaluation study, an experimental system was developed to test the accuracy of automatic critiquing using 22 JavaML critiquing rules. Twenty-eight Java programs were randomly selected from student homework submission in two introductory Java courses as test cases. Three-hundred fifty-seven critiques were generated on the programs. Our domain expert, a computer science professor, rated each critique with *good* or *inappropriate*. *Inappropriate* indicates that a critique is either wrong or not worth saying in the given context.

Among the generated critiques, 68% were rated good and 32% were rated *inappropriate* (see Figure 12). By analyzing each critiquing rule's accuracy, only three rules were found to have fault rates over 23%. One was a rule that critiques functions that contains more than one thousand characters for being too long. This critique was not appropriate in some situations such as when a function was used to construct a graphic user interface. The reason for this problem was that the system could not understand the context or purpose of the function. The other two problematic critiquing rules had incorrect patterns. After removing these three problematic rules from automatic critiquing,

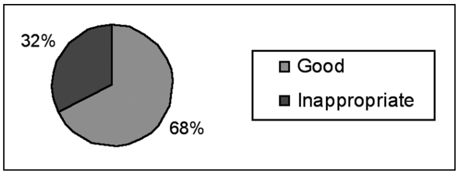


Figure 12. Ranking of 357 generated critiques

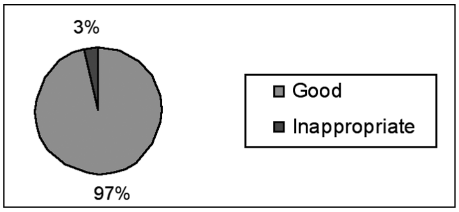


Figure 13. Ranking of 205 generated critiques without three problematic critiquing rules

the accuracy rate dramatically increased to 97% (see Figure 13). With the total number of generated critiques decreased to 205, each program still received 7 good critiques in average. This performance is satisfactory in providing useful feedback to students as well as saving effort for teachers.

The evaluation of usability for incremental authoring was done with two teachers using the Java Critiquer together for university-level introductory programming courses. The teachers used the Java Critiquer to critique Java code as well as HTML and JSP code. A total of 436 critiques were collected during the use. Of these, 232 critiques were for Java. Forty of them had regular expression patterns. Nineteen of them had JavaML patterns. Fifty-six critiques were collected for HTML. Fourteen of them had regular expression patterns.

The Java Critiquer has been extended with a Lisp parser for critiquing Lisp code. It has collected 148 Lisp critiques. Nine of them have Lisp patterns. The reason for having such a small number of patterns is that most Lisp critiques with patterns have been thoroughly tested and offered to students in a student version of the critiquer for self-assessment. Students are required to check their code with the critiquer and correct all mistakes before submitting their code to the teacher. Therefore, there is no need to have those critiques in the teacher version.

Figure 14 shows the distribution of the critiques in the current system. The large number of critiques collected suggests that the Java Critiquer has successfully supported teachers in incrementally authoring critiques.

The Java Critiquer has been used for one year and ten months. It was first used by one teacher and then shared with another teacher. Among the total of 232 Java critiques collected in the system, one teacher authored 98 and the other authored 134. During the use of the system, the number of Java critiques reached a maximum around 200 and has remained relatively stable for

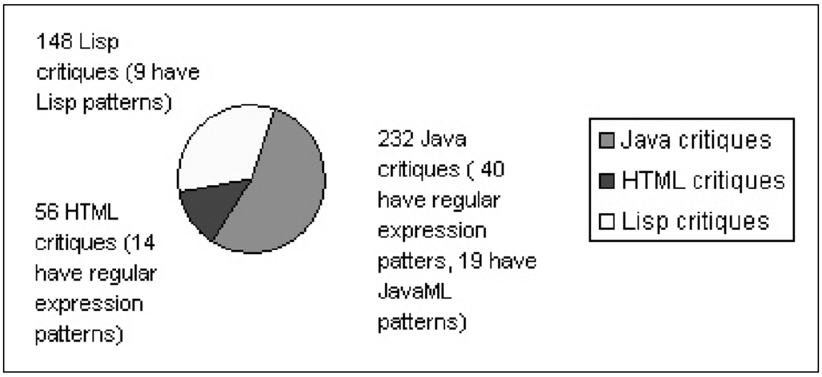


Figure 14. Critiques in the Java Critiquer

over a year with occasional new critiques to refine existing critiques. This suggests that critiques converged when the system was used by authors with similar views and skills for a fixed syllabus of topics.

When the critiques in the Java Critiquer became relatively stable, the system was introduced to several teaching assistants (TAs). The TAs were given limited access to the system in that they could add new critiques into the system, but could not modify or delete existing ones. The TAs were given a homework exercise before they started using the system to critique student code. They were asked to critique several programs where they needed to modify and delete critiques generated by automatic critiquing, as well as manually apply existing critiques and add new critiques into the system. This was to make the TAs familiar with the critiquing process and aware of the need to verify and complement the results of automatic critiquing.

When the TAs used the Java Critiquer, they tended to write new critiques duplicating existing ones in the system. Their new critiques were usually overly specialized versions of existing generalized critiques. The cause of the problem was that they did not believe the critique database was extensive enough to cover most critiques. Therefore, they did not completely master the critiques in the database beforehand. In addition, it was difficult for them to understand a general critique without seeing how the critique was applied.

Future research is needed to find ways to facilitate the sharing and use of existing critiques. Our experience with another critiquing system developed for a commercially offered programming course suggests that allowing authors to view examples of the use of a critique helps the understanding of a critique. The system records every use of a critique with the critique so that authors can easily retrieve previous use cases of a critique. The feature greatly helped one of our authors understand the critiques created by others. Another way to facilitate the use of existing critiques is to improve the interface for finding critiques. The current Java Critiquer allows the user to do a keyword search or select a name from a list to find existing critiques. Additional features such as automatically suggesting similar critiques on the side when the user types a new critique could help the user choose an existing critique in the process of writing a new one. This would save the author the effort of doing an explicit search. More studies are needed to compare and evaluate different approaches.

RELATED WORK AND DISCUSSION

In the following, we discuss work related to the Java Critiquer in the area of software development methodologies, educational critiquing systems, intelligent tutoring systems, and systems for code review.

Incremental Development Methodologies

The Java Critiquer presents an incremental authoring model for developing educational critiquing systems. The following discusses other incremental software development methodologies.

Seeding, evolutionary growth, reseeding

Seeding, evolutionary growth, reseeding (SER) is a model describing three stages (seeding, evolutionary growth, and reseeding) in the evolutionary development of software systems (Fischer, 1998; 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 the stage where the system supports user work and collects information generated by use. Reseeding is the stage where information collected during evolutionary growth is formalized and organized to support the next cycle of development. The SER model has been used in the development of domain-oriented design environments (Fischer, 1993), organizational memories (Lindstaedt, 1996), course information environments (dePaula, Fischer, & Ostwald, 2001), and open systems approaches (Fischer & Scharff, 2000; Raymond & Young, 2001). 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.

Extreme Programming

Extreme programming (XP) (Beck, 1999; Fowler, 2001) is an agile software development methodology (Boehm & Hansenm, 2001; Constantine, 2001). It reduces the upfront development effort in the traditional waterfall model (Royce, 1970) by developing systems through a series of small full-functional deliverables. Its development process is a continual integration of new functions into deliverables to implement new use cases. Both XP and our incremental authoring model try to avoid unnecessary planning and designing effort. They share some common perspectives.

Systems in the XP model are developed to meet current requirements. No effort is spent on features that are needed in the future. Our incremental authoring model enables teachers to perform authoring during the critiquing process. Authoring work such as adding a critique or creating a pattern is done to improve the current performance of the system. It has immediate effect on the system performance.

Small full-functional releases in XP are often developed for customers to use them to provide feedback for improvement. Our incremental authoring model provides a functional critiquing system for teachers to use. During the usage, teachers identify situations where the system needs to be improved. For example, when a critique is applied incorrectly, the pattern for the critique needs to be corrected. When the system does not have a critique for a mistake, a new critique needs to be added into the system. These situations are discovered through the use of the system.

XP often requires a customer continuously involved in the development process to provide system requirements. Our incremental authoring model integrates critique authoring into the critiquing process, so that by using the system for critiquing the teacher naturally involves in the development process.

Educational Critiquing Systems

A number of rule-based critiquing systems have been developed to teach programming. For example, the Lisp-Critic (Fischer, 1987) is a system for teaching Lisp programming. It matches users' code against a large set of critiquing rules. These rules look for mistakes in the code and suggest corresponding improvements (e.g., safer list operations, more advanced functions, etc.). Information such as which rules have been triggered and what functions the user is using forms a user model. The user model determines the set of rules used to check the code. There is also a visualization tool and a browser of Lisp concepts for helping the user understand the critiques. Rules in the Lisp-Critic are specified in Lisp which makes it difficult for non-programmers to author. Its domain dependent feature further makes it hard to be reused in other domains.

Different from the Lisp-Critic, the Java Critiquer can be customized to support a wider range of domains. Its rules are in XML format with regular expression or XML patterns. They can be written for critiquing sources other than Java programs. Furthermore, the architecture of the Java Critiquer allows it to use parsers in other domains to generate XML source and use the XML pattern matcher. The Java Critiquer does not have a student model to determine which critiquing rules to use. All rules are activated during critiquing. This avoids the complexity of a student model. It, however, requires a teacher in feedback loop to verify the appropriateness of the generated critiques before they are sent to the students.

Hendrikx, Olivie and Loyaerts (2002) built a system to detect novice Java programmer's misconceptions. The system uses XSLT for pattern matching. It lets users run a local client program to transfer files to the server making it possible for their system to detect misconceptions involving code in different classes. The Java Critiquer asks users to cut and paste Java code into a browser and therefore can only handle mistakes in a single class. However, using its own pattern matcher, the Java Critiquer can use rules with Java function calls embedded. This extends the range of rules that can be written and handled by the system.

To the best of our knowledge, critiquing systems developed so far are mostly closed systems. They assume a substantial set of critiques developed at design time and do not provide interfaces for instructors to modify these critiques. The Java Critique provides an interface for teachers to author critiques in the system during critiquing. It leverages human expertise to complement automatic critiquing so that the system can function without complete critiquing knowledge. The knowledge acquisition process takes place gradually during the use of the system.

The Java Critiquer implements the incremental critique authoring model in the software programming domain. We believe that critiquing should be applicable to those subject areas where the creation of text artifacts is a critical skill. That includes programming, writing, business planning, mathematical analysis, and a host of other areas. We also believe that the incremental critique authoring model can be used to help teachers collect and author critiques in those areas. For example, one of the authors has developed and used two other critiquing systems, one for communication skills in business writing for English as a Second Language (ESL), and one for goal-based scenario (GBS) (Schank, Fano, Bell, & Jona 1993) learning environment design. Figure 15 shows two example critiques used in the critiquer for learning environment design. They address problems in student's answers to the question "what mistakes do people make and why do they matter," the first question that students need to answer when they start designing a GBS learning environment. Critiques like the ones shown in the figure were gradually collected during the usage of the critiquer and generalized into a set of pitfalls that were shared with students to help them avoid common problems when they create initial designs for learning environments.

We have also implemented the incremental critique authoring model in a learning environment authoring tool called INDIE (Qiu & Riesbeck, 2005). INDIE creates learning environments where students need to solve problems by conducting simulated experiments, collecting data, generating hypotheses, and using the data to support or refute the hypotheses. INDIE provides

Answers to the question "What mistakes do people make and why do they matter?"	Critiques
Saying yes to expenditures and signing bills without thinking of the timing of when money from the budget will actually be used.	Close but "without thinking" is not a mistake, nor, for that matter, is saying yes. A mistake is something linked directly to a bad outcome, e.g., "they say yes to all requests, use up their budget in one month, and have no money left for the rest of the year." "Not thinking" is an explanation of why they make this mistake, albeit a vague one.
New managers often struggle with understanding how to maintain the monthly budget for their department.	This is vague. First, understanding is too broad a term. Even an expert might be accused by another of not really understanding something. Second, maintaining a budget is also too broad.

Figure 15. Example critiques on students' answers to the question "What mistakes do people make and why do they matter?"

a critiquing interface where teachers can critique problems in student work in the learning environment (e.g., running an unnecessary experiment or generating an unsound conclusion from inadequate data), and save the critiques for future reuse. For example, Figure 16 shows three example critiques created by a teacher when using Corrosion Investigator, a learning environment delivered by INDIE where students work as environmental engineering consultants to help a paper processing company diagnose the cause of its recurring pipe corrosion. These critiques address student's data interpretation and critical reasoning skills in diagnostic problem-solving. We are experimenting with natural language processing techniques to help teachers retrieve and reapply these critiques.

Intelligent Tutoring Systems

Intelligent tutoring systems (Wenger, 1987) employ detailed student modeling to provide individualized feedback to students. They asked a student to solve a specific problem, and analyze the student's solution to update and refine an internal model of the knowledge and misconceptions that the student has. Tutoring rules use the student model to guide the selection of feedback and future problems to pose.

The Lisp Tutor (Anderson, Conrad, & Corbett, 1989) is one of the earliest systems developed to teach Lisp programming. Students using this tutor are asked to solve programming exercises. These exercises are usually determined by the system's understanding of the student's competence level based on the interactions with the student. While the student writes programs in the system, the tutor closely monitors student's moves. It uses about 500 pro-

Student work	critique
Test Result: [Water Chemistry check point 9]SO4: 83.08 mg/L Reason: High sulfate is still present, indicating SRB's may be active.	This is NOT evidence supporting chemical corrosion as a cause.
Test Result: [Water Chemistry check point 3]pH: 6.378 Reason: Neutral pH, indicating process is probably not a chemical one	There are other possibilities for chemical corrosion at neutral pH's - should acknowledge this.
Test Result: [Water Chemistry check point 9]H2S: 32.546 mg/L Reason: Not as high as in recirculating pipes. Corrosions may be a combination of bio and chemical processes.	Not well explained- is H2S derived from activity at that location, or is it left over from water derived from flushed recirculating water.

Figure 16. Example critiques in the Corrosion Investigator learning environment

duction rules to generate the next correct moves that the student should make, a technique called model tracing (Anderson et al., 1989). If the student makes a move that differs from the predicted steps, the tutor provides feedback to indicate problems in the code, suggestions for the right function, or pseudo code for the next step. The tutor also uses a set of buggy rules to detect mistakes in student work. Using the tutor, the student receives step-by-step instructions on how to write a program. Unnecessary time and effort can be saved from debugging a simple syntax error or pondering on a wrong path. The student always stays on the right track by following the advice from the tutor. Over the years, various tutoring systems have been built to teach programming (e.g., PROUST (Johnson, 1986), MENO-II (Soloway, Rubin, Woolf, Johnson, & Bonar, 1983), and ELM-PE (Weber & Möllenberg, 1995).

Intelligent tutoring systems provide step-by-step support for completing a program, but they usually need extensive knowledge about the domain content, student modeling and pedagogical strategy (Wenger, 1987). In addition, students can only work on a predefined set of exercises in the system. Critiquing systems do not have these limitations. They can critique any code, which enables them to be beneficial for both beginner and intermediate level programmers. Furthermore, our incremental model allows authoring to be done at run-time. Systems can be developed with relatively small initial effort and be put into use early to test their efficacy. Only when the initial design proves to be effective, further authoring effort is invested. During the authoring process, instructors can receive immediate benefits from their authoring effort.

Systems for Code Review

There are existing commercial systems that provide code review for programmers. For example, LINT (Johnson, 1978) is a tool that checks C code for potential problems not caught by a compiler. It reports issues such as questionable pointer assignments, unreachable code, and unused variables. It is designed to be used on code after its successful compilation. Pattern-Lint (Sefika, Sane, & Campbell, 1996) is a similar tool but focuses on design-level problems. It checks code against its design specifications to make sure the code faithfully implements architectural models and design principles such as design patterns in its intended design. SoftBench CodeAdvisor (Hewlett-Packard Company, 1998) is a tool that provides advanced code checking for C and C++. It can detect a variety of actual and potential code problems that compilers can not detect, for example, potential heap corruption, dangling pointers, ambiguous initializations, and dependencies on system-specific compiler/linker behavior. It allows users to add custom rules by writing C++ classes using provided API (application programming interface). CodeWizard (Kolawa & Hicken, 1998) is a tool that concentrates on C++ coding standards and recommended practices. It allows user to add or customize rules used in error checking using a wizard. Checkstyle (URL: http://checkstyle.sourceforge.net) and PMD

(URL: http://pmd.sourceforge.net) are two open source tools that scan Java source code and report violations of coding standards. The above systems focus on providing feedback to software developers on memory management, bug detection, deviation from standards, and design flaws. They are not intended for educational purposes.

CONCLUSIONS

We have described a development model that allows teachers to incrementally author a critiquing system during use. Just as "design for testing" affects system design, so does "design for incremental authoring." Systems designed for incremental authoring do not use complex domain models to generate feedback. Its architecture has to provide a place for a human in the loop. That means that the system needs to be able to display inputs and system responses to authors in a readable form. It needs to allow the human 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. By providing an architecture and interface for in-task authoring, systems in our model support early deployment and testing through instructor involvement.

We described the Java Critiquer, a critiquing system that we built, as an exemplar of our model. The Java Critiquer helps teachers 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 supports critique authoring. We believe our model presents a practical and beneficial approach to developing critiquing systems for education.

References

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*, 467-505.

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.

dePaula, R., Fischer, G., & Ostwald, J. (2001). Courses as seeds: Expectations and realities. *Proceedings of the Second European Conference on Computer-Supported Collaborative Learning (Euro-CSCL' 2001)*, Maastricht, Netherlands, 2001, pp. 494-501.

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, *5*(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, 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.

Fischer, G., & McCall, R. (1989). Janus: Integrating hypertext with a knowledge-based design environment. In *Proceedings of the ACM Hypertext'89*, p. 105-117. ACM, November 1989.

Gertner, A. (1995). *Critiquing: Effective decision support in time-critical domains*. Ph.D. Dissertation, Dept. of Computer and Information Science, University 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.

Johnson, W. L. (1986). *Intention-based diagnosis of novice programming errors*. London: Pitman.

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., & Riesbeck, C. K. (2005). The design for authoring and deploying web-based interactive learning environments, In *Proceedings of World Conference on Educational Multimedia, Hypermedia & Telecommunications (ED-MEDIA)*, June 2005.

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

Schank, R., A. Fano, B. Bell, & M. Jona. (1993). The design of goal-based scenarios. *Journal for the Learning Sciences, 3*(4). 305-345.

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.

Soloway, E., Rubin, E., Woolf, B., Johnson, W. L., & Bonar, J. (1983). MENO II: An AI-based programming tutor. *Journal of Computer-Based Instruction, 10*, 20-34.

Wenger, E. (1987) *Artificial intelligence and tutoring sustems: Computational and cognitive approaches to the communication of knowledge.* Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Weber, G., & Möllenberg, A. (1995). ELM programming environment: A tutoring system for LISP beginners. In Wender, K. F., Schmalhofer, F., and Böcker, H.-D., eds., *Cognition and Computer Programming*. Norwood, NJ: Ablex Publishing Corporation, 373-408.