Facilitating Critiquing in Education: The Design and Implementation of 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. However, it is 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 tools, which offer students individualized coaching and just-in-time learning opportunities. This paper describes the Java Critiquer, a critiquing system to teach students how to write clean, maintainable and efficient code. The system uses a framework that can be customized to support educational critiquing in other domains. Preliminary results from a pilot test show that the Java Critiquer is capable of providing useful feedback to students as well as helpful support for teachers.

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; Graesser, Person, & Magliano, 1995). 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 (Pollins, 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 making personalized critiques is labor-intensive and time consuming for teachers. Students also have to wait for feedback before continuing their work. Moreover, expertise required for performing critiquing is not always available.

One way to avoid the above problems is to build software systems to critique student work. Critiquing systems have been developed in various domains, e.g., Lisp programming (Fischer, 1987), 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. With assistance from critiquing systems, students can work at their own pace but still get individualized coaching and scaffolding.

The Java Critiquer

One domain suitable for applying the critiquing paradigm is computer programming. In programming, a common misconception among novice programmers is that as long as a program runs and generates the right result, the work is complete. Such lack of understanding often leads to code with poor efficiency and readability, and also may bring bad habits into future software development. One cause of such problems is that many programming exercises simply require students write workable code, but do not emphasize the importance of writing clean,

maintainable and efficient code. Furthermore, even when good programming principles are demonstrated by teachers, it does not necessarily ensure that students know where and how to apply them during practice.

A web-based critiquing system called the Java Critiquer was developed to solve this problem. The Java Critiquer detects and critiques bad programming style. Critiques are aimed at mistakes often seen in introductory programming courses. Students in class will be required to check their code using the tool and correct all the mistakes before submitting their code. Such a procedure ensures that students not only produce correct code, but also code that is cognitively efficient (e.g., more readable and concise) and machine efficient (e.g., smaller and faster) (Fischer, 1987). A similar tool is being used in an introductory Lisp class at Northwestern University. We intended to use the Java Critiquer in the same way, to allow any novice programmer to use it for code review. A similar practice of code critiquing has been seen in the XP (eXtreme Programming) software development methodology (Beck, 1999). XP programmers work in pairs to critique each other's code and therefore ensure the quality of the code.

In the following, we will describe the architecture of the Java Critiquer, how its implementation fulfils our design goal, and preliminary results from a pilot test.

System Architecture

The Java Critiquer is built not only to solve the problem in teaching programming, but also as a framework that can be customized to support educational critiquing in other domains. To build such an extensible and reusable system, we had the following design goals in mind:

- The architecture should be general enough so that it can handle content in different domains.
- Authors should be able to customize critiques without modifying large parts of the system. Authors should also be able to associate specific critiques with specific situations.
- To be practical, the system should be able to run as a stand-alone application, but be ready to be integrated into various learning environments, e.g., goal-based scenarios (Schank et al., 1993).
- The system should be easy to access and use with little overhead for users.
- The system should focus on addressing the common and simple mistakes most novices make.
- The system should use a lightweight critiquing mechanism that does not require reasoning, planning or goal representations.

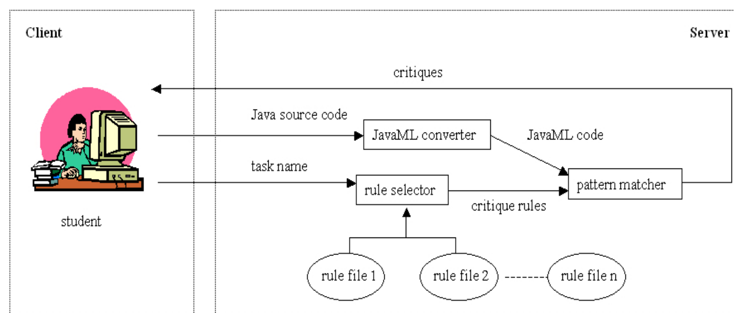

Figure 1: Architecture of the Java Critiquer.

These design goals lead to the system architecture illustrated in Figure 1. The system employs the usual web-based client-server model and uses rule-based analytical critiquing (Fischer et al., 1991).

A JSP page lets users select which programming task they are working on, e.g., "Task 1: Circle" as shown in Figure 2, or "General" which means no specific task is associated with it. Users copy their Java source code into a text area. The code and the task name are sent to the server. The Java code is translated into JavaML (Badros, 2000) by the *JavaML converter* (see Fig. 1). The task name prompts the *rule selector* to load in particular rule files for the specific task. The *pattern matcher* matches the patterns in the rules against the JavaML code, and returns a list of triggered critiques. The critiques are then inserted right below the problematic code and displayed in the JSP page (see Fig. 2). The implementation of the rule-based pattern matching framework is a variation of the LMX (Language for Mapping XML) processor (Maruyama, Tamura, & Uramoto, 1999).

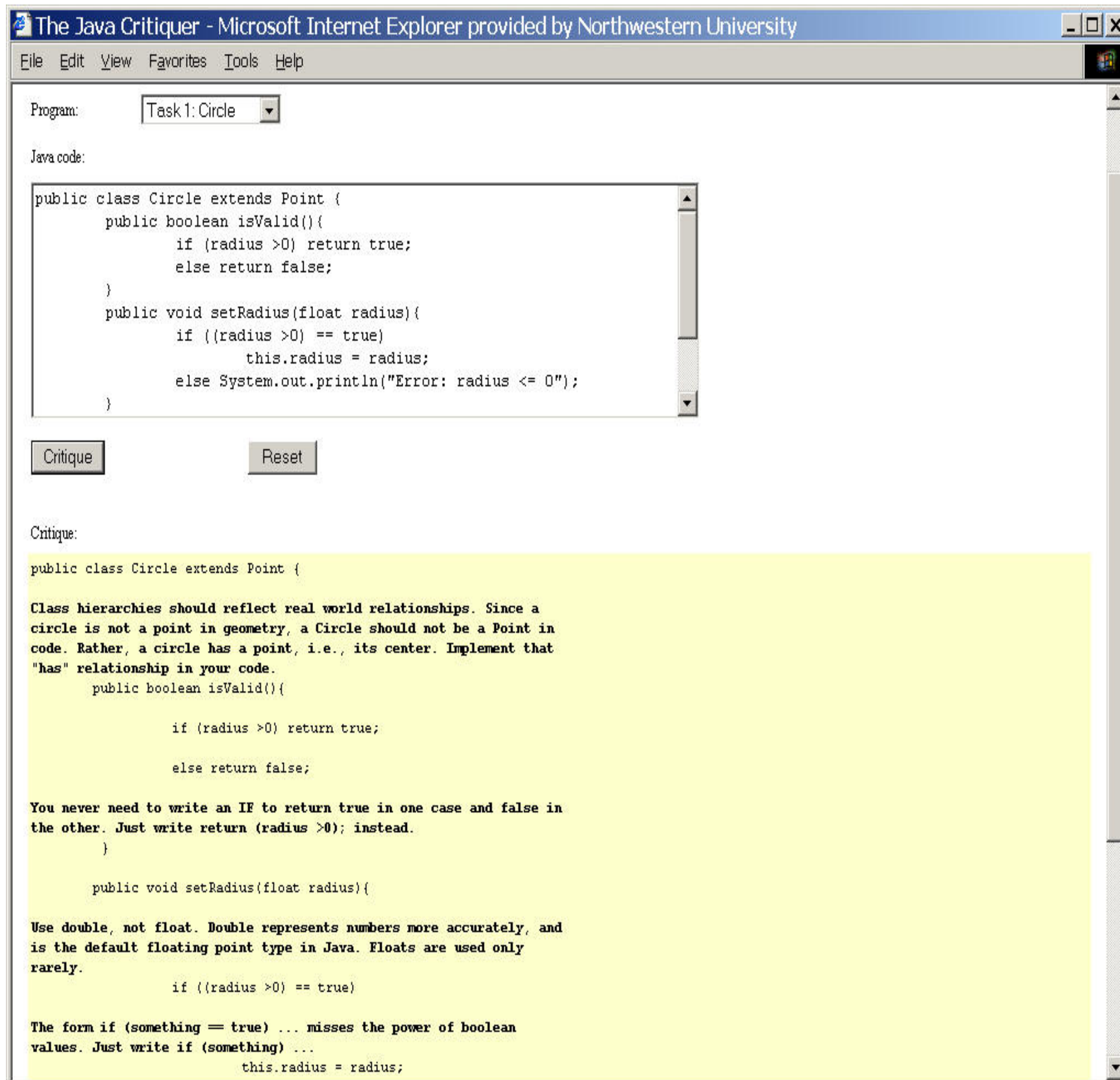

Figure 2: The Java Critiquer displays the critiques inside the user's code.

There are two strategies of critic intervention, active and passive (Fischer et al., 1993). Systems that use the active strategy continuously monitor user actions and disrupt the user as soon as a critique is triggered. In contrast, systems with the passive strategy require the user to invoke the critiquing process. Our system chooses the passive strategy over the active strategy in that we wanted users to concentrate on their tasks without intrusion, and there is no need to prevent mistakes from occurring. Allowing users to make mistakes and correct them afterwards helps users realize the utility of the knowledge and therefore provides them important opportunities for effective learning (Schank, 1999).

Critiquing Rules

Critiquing rules used by the pattern matcher are stored in XML format (Bray, Paoli, & Sperberg-MacQueen, 1998). For example, Figure 3 shows the rule that is used to generate the critique for the code shown in Figure 4. The left-hand side of the rule is a pattern to match JavaML code. The right-hand side of the rule is a critique. Both sides can have variables embedded. The variables in the patterns are bound to the values in the matched code, e.g., a chunk of JavaML code, a single JavaML element, or an attribute value of a JavaML element. The variables in the critiques are

substituted with corresponding bounded values. This allows the user's code be quoted in the critique, making the critique contexualized advice that is specific for the user's code, rather than a general one that is out of context.

```
<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 3: A critiquing rule in the Java Critiquer.

```
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.

In order to broaden the expressive capacity of the patterns, specialized Java classes are built for the logical connectives ("and", "or" and "not"). Other functionalities that are difficult to express in XML are also written in Java. For instance, a Java function is built to detect if the first character of a string is in lower case. This function is used in a pattern to detect class or interface names starting with the first character in lower-case. Such Java utility functions are not hard-coded into the pattern matcher. Instead, they are saved individually in Java files. When the *pattern matcher* detects an "extension" tag with a class name, the corresponding Java file is loaded during runtime using Java reflection API. This mechanism allows rule authors to make their own Java utility functions and utilize them in patterns without changing the *pattern matcher*. This feature enables the system to accommodate additional critiques to address other errors.

A default rule file contains the generic critiques that can be applied to all Java code, e.g., "The form ‘if (something == true)…’ misses the power of Boolean values. Just write if (something) …” Other rule files contain specific critiques, that are only applicable to specific tasks. For instance, a critique for the circle exercise is "Class hierarchies should reflect real world relationships. Since a circle is not a point in geometry, the Circle class should not be a subclass of the Point class in code. Rather, a circle has a point, i.e., its center." The default rule file is used in any situation. Specific rule files are loaded by the rule selector according to specific programming tasks.

Critiques from the Java Critiquer usually include a description, the reason for the critique, the context of the critique and directions for improvement. These elements are important for education. Learners should be told the reason for the critique rather than simply being told what to do (Gertner, 1995).

The Pattern Matcher

The pattern matcher takes an XML tree and a list of patterns as its input. Starting from the root of the tree, it traverses the tree and matches every pattern against every node in the tree. When a pattern is matched to a JavaML code, the corresponding critique is instantiated and saved into a list. The pattern matcher is also responsible for dynamically loading Java utility functions used in patterns.

The pattern matcher does its work without knowing anything specific about JavaML. It can take any XML source as input, i.e., MathML, UML. We have been experimenting in the math domain to critique students' mathematical proofs. A Critiquer for UML modeling is also under development employing the same framework. The domain independent feature of the matcher makes the system suitable for creating additional critiquers for other domains.

Evaluation

The goal of our pilot test was to access the quality of critiques generated by the system and identify opportunities for improvement. We used the Java Critiquer on 28 programs selected randomly from student homework submission 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.

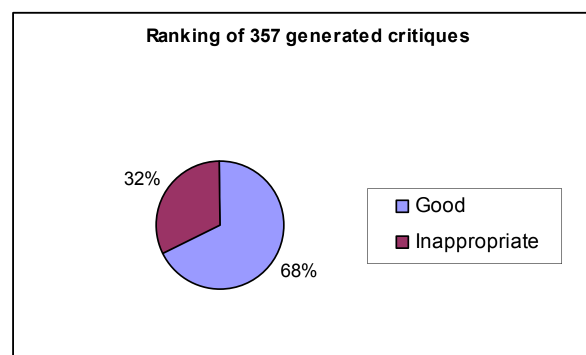

Figure 5: Ranking of 357 generated critiques.

Among the total of 357 critiques that were generated, 68% were "Good" and 32% were "Inappropriate" (Fig. 5). Such accuracy rate was not satisfactory for employing the system in real-world situations. By analyzing each individual critiquing rule's accuracy rate, however, we found that among the total 22 critiquing rules in the system, only three had fault rates over 23%. One was a critiquing rule applied to functions containing more than one thousand characters, saying they were too long. This was not applicable in some situations, e.g., functions that construct a graphic user interface. This failure was caused by not being able to recognize the context of the program. The failure of the other two critiquing rules was due to incorrect patterns. After removing these three problematic critiquing rules, the accuracy of the system dramatically increased to 97% (see Fig. 6). With the total number of generated critiques decreased to 205, in average, each program still received 7 valid critiques. With such performance, we believe the use

of the system can be helpful for providing feedback to students as well as saving work for teachers.

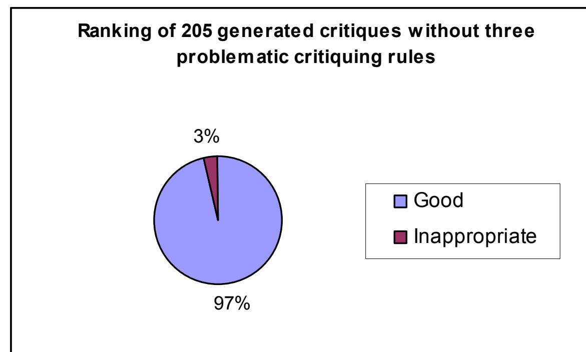

Figure 6: Ranking of 205 generated critiques without three problematic critiquing rules.

Besides refining individual critiquing rules to improve the quality of system generated critiques, we are also integrating our system with a manual critiquing interface. For those problematic or hard to implement critiques, this interface allows teachers to apply them manually or verify them after computer generation. This approach ensures the quality of critiques by leveraging human expertise to complement the feedback generated from the system. Additionally, it allows teachers to collect critiques for later use or sharing, avoiding the difficulty of generating all the critiques upfront.

Discussion and Future Work

Hendrikx, Olivie and Loyaerts (2002) built a system to detect novice Java programmer's misconceptions. The major difference between their system and the Java Critiquer is that Hendrikx's system uses XSLT for pattern matching whereas the Java Critiquer uses its own pattern matcher. With a specialized pattern matcher, the Java Critiquer uses rules expressed in both XML format and Java code, which expands the capability and applicability of the system. Hendrikx's system lets users run a local server program to transfer files to the server making it possible for their system to detect misconceptions involving code in different classes. Our system allows users to cut and paste their Java code into a browser and therefore can only handle misconceptions in a single class.

The Lisp-Critic (Fischer, 1987) is a similar tool to help users improve their Lisp code. Its rules, however, are specified in Lisp making it difficult for non-programmers to author. Its domain dependent feature, moreover, makes it hard to be reused in other domains.

There are existing tools that provide feedback to programmers on their code, e.g., LINT (Johnson, 1978), Pattern-Lint (Sefika, Sane, & Campbell, 1996), CodeAdvisor (Hewlett-Packard Company, 1998), CodeWizard (Kolawa & Hicken, 1998). They usually focus on bug detection, memory management, coding standards, and design flaws. They are not for educational purposes and are usually tied to particular development environments.

The function of our pattern matcher is similar to PatML (see IBM Alpha Works). PatML, however, is currently under a limited-time license agreement with no source code available.

Current critiquing rules in our system were made by modifying the corresponding JavaML code of the problematic Java code. Compared to Java, JavaML is relatively hard for Java experts to work with. We have been developing an authoring tool that facilitates authors to write rules directly from the problematic Java code. Authors can embed pattern variables in Java code and the tool will generate the JavaML pattern accordingly.

Conclusion

In summary, we have described the Java Critiquer, a critiquing system to teach students how to write clean, maintainable and efficient code. The system uses a framework that can be customized to support educational critiquing in other domains. The principle goal of building such systems is to provide individualized feedback and just-in-time learning opportunities to students. Results from a pilot test show that the Java Critiquer is capable of providing useful feedback to students as well as helpful support for teachers.

References

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

Beck, K. (1999). *eXtreme Programming eXplained: Embrace Change*. Addison-Wesley.

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.

Bray, T., Paoli, J., & Sperberg-McQueen, C. M. (1998). Extensible markup language (XML) 1.0. *W3C Recommendation, February 1998*. http://www.w3.org/TR/REC-xml.

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," *Proceedings of the 10th International Joint Conference on Artificial Intelligenc*e, Milan, Italy.

Fischer, G., Lemke, A. C., Mastaglio, T., & Morch, A. (1991). The Role of Critiquing in Cooperative Problem Solving. *ACM Transactions on I nform ation Systems*, Vol. 9, No. 2, 1991, pp. 1 2 3-151.

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

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

Graesser, A. C., Person, N. K., & Magliano, J.P. (1995). Collaborative dialog patterns in naturalistic one-on-one tutoring. *Applied Cognitive Psychology*, 9, 359-387.

Hendrikx, K., Olivie, H., & Loyaerts, L., (2002). A System to Help Detect and Remediate Novice Programmer's Misconceptions. *Proceedings of ED-MEDIA 2002 (World Conference on Educational Multimedia, Hypermedia & Telecommunications).*

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

IBM AlphaWorks. PatML. URL: http://www.alphaworks.ibm.com/formula/patml

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

Kolawa, A., & Hicken, A. (1998). Programming Effectively in C++. ParaSoft Corporation. URL: http://www.parasoft.com/products/wizard/cplus/papers/tech.htm

Maruyama, H., Tamura., K., & Uramoto, N. (1999). *XML and Java: Developing Web Applications*. Reading, MA: Addison Wesley Longman.

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.

Schank, R. C. (1999). *Dynamic Memory Revisited.* Cambridge University Press: New York.

Schank, R., Fano, A., Bell, B., & Jona, M. (1993). The Design of Goal-Based Scenarios. *Journal of 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. *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.

Acknowledgements

We would like to thank Paul Kleczka and Joshua Ochs who helped to implement the current design, Baba Kofi Weusi-Puryear who developed the modules for the Math Critiquer, and Jonathan Arme who developed the modules for the UML Critiquer.