Web Compiler Service for Interactive Java Tutorials

Roxanne L. Canosa, Axel T. Schreiner, Dustin E. Mulcahey
Rochester Institute of Technology
134 Lomb Memorial Drive
Rochester, New York 14613
(585) 475-5810
{rlc, ats, dem5302}@cs.rit.edu

ABSTRACT
A highly interactive electronic "textbook" was created and used as a learning aid in an introductory computer science course. The textbook consisted of a set of web-based tutorials that were developed with the intent of helping students understand the *process* of programming, along with helping them to learn language syntax and program structure. The tutorials were built around a web server that compiled Java programs and returned the result to the client as an applet. The tutorials included an editing environment that allowed students to test programs and experiment with code examples embedded in the text, without the need to install or run the Java SDK on the client machine. The service is also useful for enabling rapid prototyping with Java-related compiler tools without the need to install those tools first. This paper describes the architecture of the web compilation service, explains how it was used as a learning aid in an introductory programming course, and determines its effectiveness.

Categories and Subject Descriptors
K3.1 [**Computers & Education**]: Computer Uses in Education – *computer assisted instruction.*

K3.2 [**Computers & Education**]: Computer and Information Science Education – *computer science education, information science education*.

D3.4 [**Programming Languages**]: Processors – *translator writing systems and compiler generators*.

General Terms
Languages, Experimentation.

Keywords
Introductory Computer Science Courses, Interactive Java Tutorials, Web Compilation, Pedagogy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'07, March 7-10, 2007, Covington, Kentucky, USA.
Copyright 2007 ACM X-XXXXX-XX-X/XX/X…$5.00.

1. INTRODUCTION
Presenting basic concepts in an introductory computer science course is usually accomplished by framing the concepts in concrete terms via a programming environment, such as Java. In most cases, students initially learn computer science by studying code examples from a textbook or from lecture examples, and develop programming skills by writing code via exercises, labs, and projects.

Many different classroom techniques can be used to show students examples of programs, including writing code on the board, displaying an overhead transparency, or referring to examples given in a textbook. A disadvantage of these approaches is that the examples are given in a static context with no way for the students to actually view the *process* under which the program was developed. The examples shown are complete, work as intended, and have no errors. This approach tends to give beginning students an unrealistic impression that software development is done in a single step – from initial analysis of the problem to full formulation of the solution. In addition, errors are viewed as being unusual and unexpected. This is a misleading perception, and may contribute to the difficulties students have later on with formulating software solutions to their own, more complex problems [1, 4, 5]. It also reinforces the idea that only weak programmers encounter errors during the development process, or have any difficulty arriving at the final solution once the process has begun.

Another classroom technique is to use an interactive approach. For example, code can be written "live" using a laptop connected to a projector, or the programming process can be recorded onto video and shown later to the class [1]. An increasingly popular technique is to use a visualization tool such as BlueJ [2], which combines UML-like pictures with the interactive capabilities of an IDE. While these techniques are not static, they are either limited to the classroom environment, where students passively watch the instructor write the code, or the student must install and configure a tool and learn the particular environment of the IDE.

A third approach is to provide a service to students that enables them to experiment with code outside of the classroom, in an environment that does not require any special configuration or the learning of a new tool. An ideal environment would be an editable web page that "embeds" compilation and execution within the page, surrounds the example program with text that explains how and why the program works, and allows students to easily modify the program and view the results.

The Web Compiler Service *wcs* [3] was developed and used to create this kind of environment for introductory students. The environment includes a set of web-based interactive Java tutorials, which were evaluated for effectiveness in terms of

their ability to impart basic Java programming and language concepts to novice programmers. *Wcs* has also been used successfully in upper-level courses on language processing, programming language concepts, and compiler construction [6].

2. THE WEB COMPILER SERVICE

Wcs implements a Java-based compiler on a web server and executes the code in an applet. All of the software development tools reside on the server; the client only needs to execute unsigned Java applets. *Wcs* accepts source text for small Java programs and compiles the source. The resulting class files are merged with runtime support, archived, and sent to the client as an applet, where the program is executed.

The service is implemented as an HTTP servlet which processes GET and POST requests. The request parameters are designed to fit HTML `form` elements. The requests originate from the client side via a web page with one or more `textarea` elements that serve as a trivial editing environment for programs. The tutorials consist of a set of web pages structured around initialized `textareas` that contain Java source code. The text on the web page is a detailed discussion of a selected topic, and suggests changes that the student can make to the source before submitting the program for compilation and execution.

2.1 Requests

A service is requested through a set of key/value pairs that can be specified in any order. The keys are used to select a processor, provide source text, select a format for the text, and set processing options. Some keys are grouped by a numerical suffix so that several compilation and preprocessing steps can be combined in one executable program. All keys are described in [6]. Figure 1 shows an example of how a request can be embedded in an HTML document. Figure 2 shows the web page that is generated from the given HTML code.

```
<h3>Let us begin.</h3>
<form method="post" target="_blank"
action="http://wcs.cs.rit.edu:8080/wcs/wcs">
<input value="applet" type="hidden" name="sink.0"
/>
<input value="javac" type="hidden"
name="processor.0" />
<textarea cols="80" rows="5" style="height:200;
width:100%;font:100% monospace" name="source.0">
 public class FirstClass {
  public static void main(String [] args) {
     System.out.println("This is an example
            of a simple Java program.");
       }
    }
</textarea>
<input type="submit" value="run" />
</form>
<p>This is a Java program. Click on the "run"
button. Another window will pop up, and you
should hit the "start" button in that window.
</p>
```

Figure 1. HTML code.

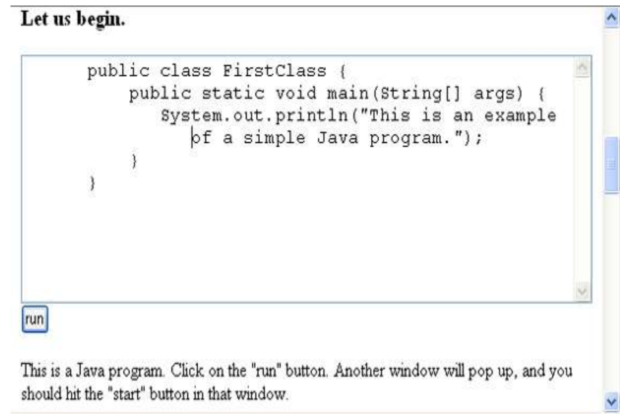

Figure 2. Edit page.

2.2 Compilation

The *wcs* servlet is executed in a Java-based container such as *tomcat* [7], and implemented using the Java `Runtime` and `Process` classes. Input and output is handled as strings, which are stored in temporary files, with the results archived in a *jar* file. Once the compilation is completed, an applet page is sent to the client browser, which subsequently asks the server for the applet's archive. At this point the servlet returns the *jar* file and deletes it from the server. While this approach makes it impossible to reload the applet page, it eliminates the need for garbage collection among files on the server. Figure 3 shows an example applet page from which the Java program can be executed.

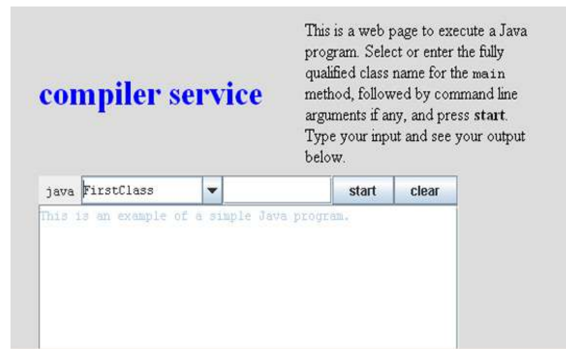

Figure 3. Applet page.

It is possible to request compilation for multiple classes and several `main` methods. If one of several `mains` is requested, the applet will contain a drop-down list, allowing the user to specify the appropriate executable. Also, note from Figure 3 that a command line argument can be specified at the top of the applet. The **`start`** button doubles as a **`stop`** button when the applet is running and can be used to curtail a runaway situation. In extreme cases the browser window may have to be closed.

2.3 Execution

The applet contains a text area that is connected with a pseudo-terminal simulation of `System.in`, `.out`, and `.err`. This would normally require signing the applet and a policy file at the client. However, a programmer is not likely to fully qualify

a reference to `System`. Therefore, the compilation process inserts a `System` class into every package it finds and all these classes delegate to the applet implementation. This is usually sufficient to support typical console input and output without a need for client-side preparations.

Program execution could pose a security problem, but applet execution always happens on the client and in a sandbox that normally prevents any access to the local platform and allows network connections only to the host of the applet. Preprocessing and compilation options are limited so that a client should not be able to cause damage to the server (other than some denial of service if huge compilations are requested). Conversely, even if *wcs* sent malicious code within the applet, the sandbox should be able to protect the client.

Some things cannot be faithfully simulated in an applet: the sandbox limits what kind of networking experiments are possible, and without additional permissions there can be no file operations. Swing can be used, but windows may stay behind until the browser window is closed – there appears to be no way to terminate platform-specific background threads such as those involved in Swing operations.

A more serious drawback concerns `static` initializations: this happens in a Java program immediately prior to the call to `main` whenever the program is executed. Currently the applet does not use a class loader and only loads classes once, some before the **`start`** button is pressed. As a consequence, `static` items are not reinitialized for subsequent **`start`** operations and output from `static` operations is not captured into the text area. One could argue that programs relying on this behavior tend to be in bad style.

3. INTERACTIVE TUTORIALS

The initial proof of concept for *wcs* was to implement a fairly extensive tutorial on language processing [6] which has been used successfully in courses on programming language concepts and compiler construction.

More recently, *wcs* was used as a learning aid in an introductory computer science course via a set of interactive web-based tutorials. The complete set of tutorials can be found at [8]. The tutorials are intended to help students who are new to programming learn basic Java syntax, program structure, programming methodology, and problem-solving techniques.

3.1 Topics

The following is a list of the topics covered by the tutorial, given in the order they were introduced to the class:

- Programming Language Basics
- Data Types in Java
- All About References
- Conditional Expressions
- Repetition
- Method Invocation
- Modifiers
- Arrays
- Software Design and Development

Advanced topics such as inheritance, interfaces, collections, exceptions, and event-driven programming may be included in future tutorial releases.

3.2 Example Tutorial

The following is a description of a complete tutorial – the first *wcs* tutorial that the students were exposed to. The topic of this tutorial is programming language basics. The complete text of this interactive tutorial can be found at [9].

The tutorial begins by comparing a programming language to a natural language such as English, explaining the similarities and differences. The objective is to enable students to make a connection from something they are already comfortable and knowledgeable about (communication using English) to something that may be new to them (communication using a programming language).

The students then see the first example, a simple Java program embedded in a form on the web page. Students are then instructed to click on the **`run`** button just below the form. This will cause an applet to come up in another window, where clicking the **`start`** button will execute the code.

Next, the tutorial explains in detail what just happened, and how the different parts of the program are related to the output that was produced. For example, the tutorial explains what a statement is and how `System.out.println()` causes something to be output to the screen. Blocks are also introduced, with attention directed to the braces used to delineate a block of code, and the label of the block. This leads to a discussion of methods and method names.

The tutorial finishes with an analogy. The syntax, structure, and organization of programs written in Java are compared to the syntax, structure and organization of literature written in English. Classes are like essays, methods are like paragraphs, statements are like sentences, and separators are like punctuation. This analogy is then connected to the concept of Java source code as *high-level* computer communication – it is meant to be easy for us as humans to read, write, and understand. Finally, compilation is explained as the process that converts the high-level code into a form that the machine can understand.

The tutorial also contains the following three exercises:

1. Change the statement that is printed to the screen to "I love Java, and Computer Science 1 is pretty cool, too." You can make the change by editing the text in the code box, and then clicking "run".
2. Write another method inside FirstClass called `public static void myMethod()`that contains a single statement that prints the line "Inside myMethod." Run the program again.
3. Add another statement to `public static void main(String[] args)` that will print the line you wrote in Exercise 2. This statement must not be another `System.out.println()` statement. Run the program once again to verify the program works as expected.

The exercises are deliberately meant to be vague. This encouraged the students to try out a number of different approaches for the exercises.

A non-interactive version of the tutorials was developed to determine how effective the interaction is with helping the

students learn programming concepts and techniques. For the non-interactive version, the text and exercises were identical to the interactive version, except that instead of active commands such as "change", "write", or "add", the exercises asked students how they would go about accomplishing the same goal.

4. EVALUATION

An informal study was conducted to gauge the effectiveness of the tutorials as a learning aid. A total of 35 students participated in the study, all freshmen taking Computer Science 1. Two versions of the tutorials were developed. One version used *wcs* to supplement the text with an interactive demonstration of the topic, and another version did not use *wcs* but provided the same textual information about the topic along with static source code for examples. Both versions of the tutorials were identical, except that the second version did not allow interaction.

Students were randomly assigned to one of the versions at the start of the study, with an approximately equal number of students assigned to each version (19 interactive and 16 non-interactive). Students were instructed to visit the website for their assigned version each week, when a new topic was posted. The hypothesis for the study is that students who learn techniques and concepts from the interactive tutorials have a better understanding of those concepts and will display a higher level of skill when applying those concepts than those who learn from the same textual explanation combined with static examples.

4.1 Weekly Quiz

Quantitative data was collected in the form of a weekly quiz that covered key concepts from that week's tutorial. Each quiz consisted of two questions. The first question involved either writing a Java code segment or predicting the output of a given code segment. The second question was conceptual in nature, i.e., it was designed to test basic understanding of a key concept presented in the textual portion of the tutorial.

All quizzes were graded by a hired grader – an upper division student who had successfully completed the course earlier and who was not aware of the nature or purpose of the experiment. The quiz grades counted for 5% of the final course grade for each student. For ethical reasons, the final quiz grade was adjusted if a significant difference was found between the average grades of the two groups. Quiz 1 consisted of the following two questions:

1. The following Java program will print to the screen "There is a quiz today." Modify this program so that the same message will be printed, but without using a `println()` statement in the `main()` method.

```
public class Quiz 1 {
   public static void main(String [] args) {
      System.out.println("There is a quiz 
today.");
   } }
```

2. What are two differences between a natural language, such as English, and a programming language, such as Java? What are two similarities?

For question 1, students were expected to write a separate method to print the message, and then call that method from the `main()` method. For question 2, students were expected to understand that programming statements must be computable, and that a programming language has less expressive power (less sensitivity to context) than a natural language, yet both have syntax and semantics, and can be used for communicating ideas.

4.2 Results

Figure 4 shows the relative distribution of quiz grades for the first quiz for both groups of students. One third of the students who accessed the interactive tutorial scored a perfect grade of 10/10 on the first quiz, and 8 students (nearly half) scored a 9 or a 10. On the other hand, one third of the students who accessed the non-interactive tutorial scored a 4/10, and only 2 students scored a 9 or a 10. It is interesting to note that the other half (9 students) who accessed the interactive tutorial scored a 4 or lower on the quiz. It appears that the interactive students either "got it" or did not "get it", whereas, the non-interactive students had a more evenly distributed quiz grade.

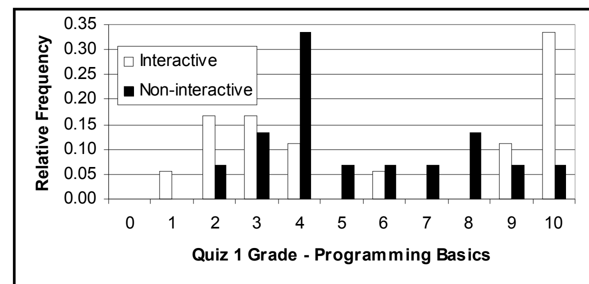

Figure 4: Quiz grades for students who accessed the interactive tutorial (white bar) and those who accessed the non-interactive tutorial (black bar).

The informal nature of this study must be stressed; it is possible that the somewhat bimodal distribution of the interactive quiz grades is merely a coincidence. However, it is also possible that some of the students who interacted with the tutorial were distracted by the possibilities for experimentation, and ignored the non-interactive (conceptual) parts of the text.

Since the quiz consisted of 2 questions, each question could be scored separately to give a separate measure of the students' understanding of the material. Question 1 (worth a maximum of six points) measured a student's programming skill, whereas question 2 (worth a maximum of four points) measured a student's retention of conceptual material. The content of question 2 was always such that the student would not know the answer based on material covered in lectures; he or she must have learned it from the tutorials or from some other source. Figures 5 and 6 show the quiz grades for the two questions.

Figure 5 shows that the students who accessed the interactive tutorial scored higher in general on the programming skill question from the quiz than did the students who accessed the non-interactive tutorial. On the other hand, Figure 6 show that the students who accessed the interactive tutorial performed *worse* on the conceptual material than did the students who accessed the non-interactive tutorial. One explanation is that the interactive nature of the tutorial may have proved too distracting for those students, who may have been spending too much time experimenting with code and not paying enough attention to the conceptual material. A more detailed analysis with a larger number of students and more tutorials is required

to come to a conclusion about imparting conceptual material using interactive tutorials.

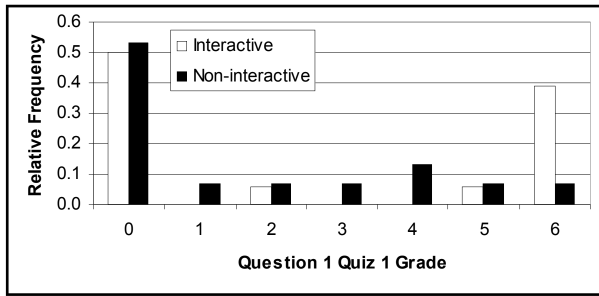

Figure 5. Quiz grades for question 1 from quiz 1.

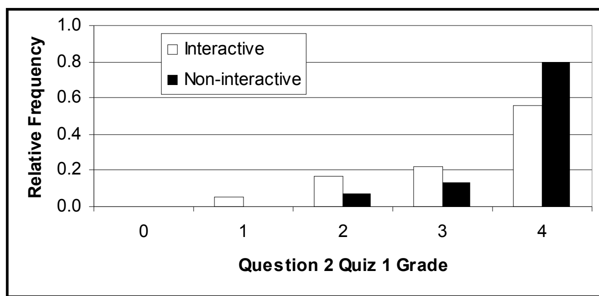

Figure 6. Quiz grades for question 2 from quiz 1.

Figure 7 shows the distribution of quiz grades for the two groups for all nine tutorials. The interactive group scored more 9s and 10s overall, similar to the quiz 1 results. However, the distribution is no longer bimodal for that group. It may be that students had become less distracted by the experimental nature of the tutorials towards the end of the term as they became more accustomed to the interaction, and this allowed them to retain more of the conceptual part of the text.

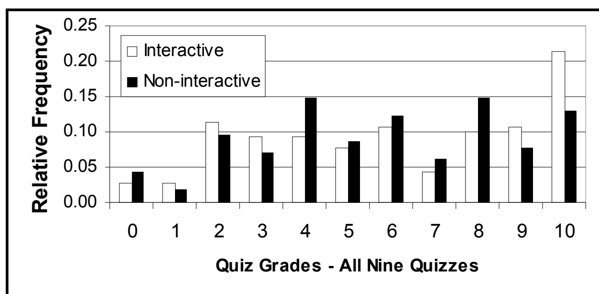

Figure 7: Quiz grades for all nine quizzes.

Also, it should be noted that some students felt less inclined to visit the website toward the end of the term (and admitted to this), and decided to answer the quiz questions based solely from lecture material and guessing. On the other hand, many of the students who were not in the interactive group confided to the instructor that they ignored their designated group and chose to visit the interactive tutorials instead. This behavior casts doubt on the validity of the results shown in Figure 7 in terms of the overall effectiveness of the tutorials over an entire term. However, the many positive comments about the tutorials from the students who switched themselves into the interactive group supports the conclusion that students enjoy this kind of learning experience, and would choose it over static content if it were available.

5. CONCLUSION

The web compiler service was used to support interactive, web-based tutorials for students who are new to the programming process. Thirty-five students from an introductory computer science course participated in an informal study on the effectiveness of the service for teaching Java programming skills and basic language concepts. The study provided evidence that students enjoy being able to experiment with code that is embedded in a tutorial explanation of the concepts and techniques, and perform better on quizzes testing programming skills. This suggests that this kind of learning aid is useful for imparting programming skills and engaging students in the programming process. On the other hand, students may become overly eager to experiment in such an environment, and may not take full advantage of conceptual material that is presented alongside an interactive example. This suggests that caution should be exercised when conceptual material is presented alongside interactive demonstrations, as students may ignore much of the content if they are busy experimenting with code.

6. ACKNOWLEDGEMENT

The development of *wcs* was supported by a Provost's Learning Innovation Grant from the Rochester Institute of Technology.

7. REFERENCES

[1] Bennedsen, J., and Caspersen, M.E.: "Revealing the Programming Process", *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 2005 pp. 186-190.

[2] Kölling, M., & Rosenberg, J.: "An object-oriented program development environment for the first programming course", SIGCSE Bulletin 28(1), 1996, pp. 83-87.

[3] Schreiner, A.T., Canosa, R.L., & Mulcahey, D.H: "Web compiler service", poster presented at the IEEE Upstate New York Workshop on Communications and Networks, Rochester Institute of Technology, November 18th, 2005.

[4] Soloway, E.: "Learning to Program = Learning to Construct Mechanisms and Explanations", *Communications of the ACM*, 29(9), 1986, pp. 850-858.

[5] Spohrer, J., & Soloway, E.: "Novice Mistakes: Are the Folk Wisdoms Correct?", *Communications of the ACM*, 29 (7), 1986, pp. 624-632.

[6] http://www.cs.rit.edu/~ats/projects/lp/doc/wcs/package-summary.html

[7] http://www.jakarta.apache.org/tomcat/

[8] http://www.cs.rit.edu/~rlc/Red/

[9] http://www.cs.rit.edu/~rlc/Red/basics.tutorial.html