An Online Programming Assessment Tool

Graham H.B. Roberts and Janet L.M. Verbyla

Embedded Systems Laboratory and
School of Informatics and Engineering
Flinders University of South Australia
PO Box 2100, Adelaide 5001, South Australia

{graham,janet}@infoeng.flinders.edu.au

Abstract

The role of assessment in modern university curricula has become far more diverse and open to scrutiny in recent years. Although its most significant role is as a measure of a student's knowledge and skills, the role as a learning device has become increasingly important and as a consequence informative and useful feed back is critical to achieve good learning outcomes

This paper describes a tool that provides a self-contained, easy to use, programming environment that facilitates the development, testing and marking of programming tasks in addition to the presentation and marking of "standard" examination questions. The tool takes responsibility for many of the tasks that program development normally requires so that the student can focus on the task of writing program source code. It has been used in a Java programming topic for two consecutive years.

Keywords: Java, Online Assessment

1 Introduction

The motivation for this work comes from both a response to the need for more efficient, effective and flexible assessment procedures and a belief that assessment, and in particular examinations, should approximate the context in which the activity would normally be performed. This is supported by complaints from students with respect to examinations that required the writing of code segments where they did not have access to the usual programming related information and tools (for example, API documentation).

Although a lot of effort has been put into the study of varying forms of online assessment, very little has been done on environments that support automated assessment of the ability to write and debug programs. The most common mode of assessment is through assignment work where demonstrators and tutors give students the opportunity to improve their programming skills and knowledge in a supportive environment. *Pairing* can act to enhance the learning outcome even more (Williams and Kessler 2000).

When assessment is the only goal, one of the most expedient methods is through multiple-choice examination such as Sun's Java Certification Examination or Novell's certification examinations. In courses where one of the primary goals is to produce competent programmers, then it is reasonable to expect examinations to contain programming tasks that can be solved in an environment that is similar to the "normal" program development environment.

2 Related Work

The teaching of introductory programming topics has been one of the most studied and contentious areas of Computer Science education since the field became an identifiable discipline. The assessment of students' ability to program is a similarly difficult and contentious issue (for example, see MCCRAKEN 2001).

Systems like WebCT (WebCT 2002) provide a range of question types, such as, multiple-choice, matching, calculated etc. While it is typically possible to specify patterns to define correct answers, this approach is not generally extensible to testing the correctness of answers to programming problems.

The system developed by Preston and Shackelford (1999) seeks to improve the ease with which assessors can mark assignments and provide feedback and to also automate some of the processes involved. While their system has some goals in common with ours, its focus is clearly on post-submission processing and not on the presentation and automated feedback required for an online programming examination

Jackson (2000) uses a composite approach to the assessment of assignment work, including applying style metrics to student programs and allowing tutor input into the process. Again, the focus is on post-submission assessment and not an interactive examination environment. Similar comments apply to Joy and Luck (1998) and Jackson and Usher (1997), just to name a few.

3 The Tool

The examination tool we have developed is a client/server application with "fat" clients and "thin" servers. While the system could have been designed with thinner clients, reasons of security and load distribution have dictated its structure.

3.1 The Server

While typically there need only be one examination server running, in an environment with several physical servers it may be beneficial to run a server on each machine, particularly if student processes are also running on those machines. Consequently, when a client is started it looks for a server, first on the local host machine, and then on a configurable list of machines.

A server application is responsible for:

- User name and password checking - a client's user name and password are checked against a list contained in a password file.
- Connection management - establish and close socket connections with clients
- Storing the time (on disk) for which a client has been connected, issuing warnings when there is only a small amount of time left, and disconnecting clients, if necessary, at the end of the exam
- Delivering questions to the client in response to a request
- Comparing the output of program execution with the expected output and delivering the results of that comparison to the client
- Storing each client's answers (on disk).
- Creating a log file of client connections and other information.
- Marking clients' answers and recording their scores for each question.

A number of scripts provide other server related functionality. For example, once students have completed their examinations, a (Unix specific) script can be run that emails them their answers, marks, the data their programs were tested on, the output from their programs and solutions to all questions. Future versions of the server will incorporate this functionality into platform independent server code.

All servers store answers in the same directory and so, for example, if a student is disconnected (for example, if the machine their client is running goes down or for some other reason their session needs to be restarted) they can reconnect to any server and their stored answers will be available to them. Further, their clock is not stopped but an administrator can modify the timer to make allowance for missing time. There is also a facility to allocate different total examination times for different classes of students (for example, students whose first language is not English can be given extra time).

The server has a simple GUI interface that displays a list of the user IDs of clients that are connected and buttons to exit, reload the password file and to mark all examinations (see figure1).

Figure 1: Server GUI

3.2 Clients

The aim of the design of the client was to provide an environment in which student can focus on developing solutions rather than on the mechanics of writing, compiling and testing programs. It is a type of Integrated Development Environment (IDE) that has inbuilt test stubs and test input data sets that takes a pessimistic view of system reliability (it saves program source code whenever the client initiates testing or changes to a different question).

The client application is responsible for:

- Attempting to connect to a server by searching a list of machines on which a server may be running. The first server that is willing to accept a connection will be sent a user name and password.
- Allowing answers to be entered and sent to the server.
- Allowing the font size to be increased or decreased
- Allowing the selection of questions.
- Allowing either *jikes* (IBM's native Java compiler - see Jikes 2002) or *javac* (Sun's standard Java compiler) to be selected.
- Displaying the amount of time remaining to complete the exam and to give warnings when end of the exam is near.
- Displaying the status of each question (visited or not, answered or not, errors, correct).
- Allowing the testing of programs on user-supplied test data (rather than the standard test data).
- Reverting to the original answer provided for the question.
- Reverting to the last version of the answer (that resulted from saving the source file, testing or change of question).

A client application has two primary windows:

1. A question/control/answer window that is divided into four areas (see figures 2 and 3):
    a. A fixed height area containing a timer that gives the time remaining in hours and minutes.
    b. A variable height question display area.
    c. A fixed height menu bar that is comprised of six click-buttons and two drop down selection menus.

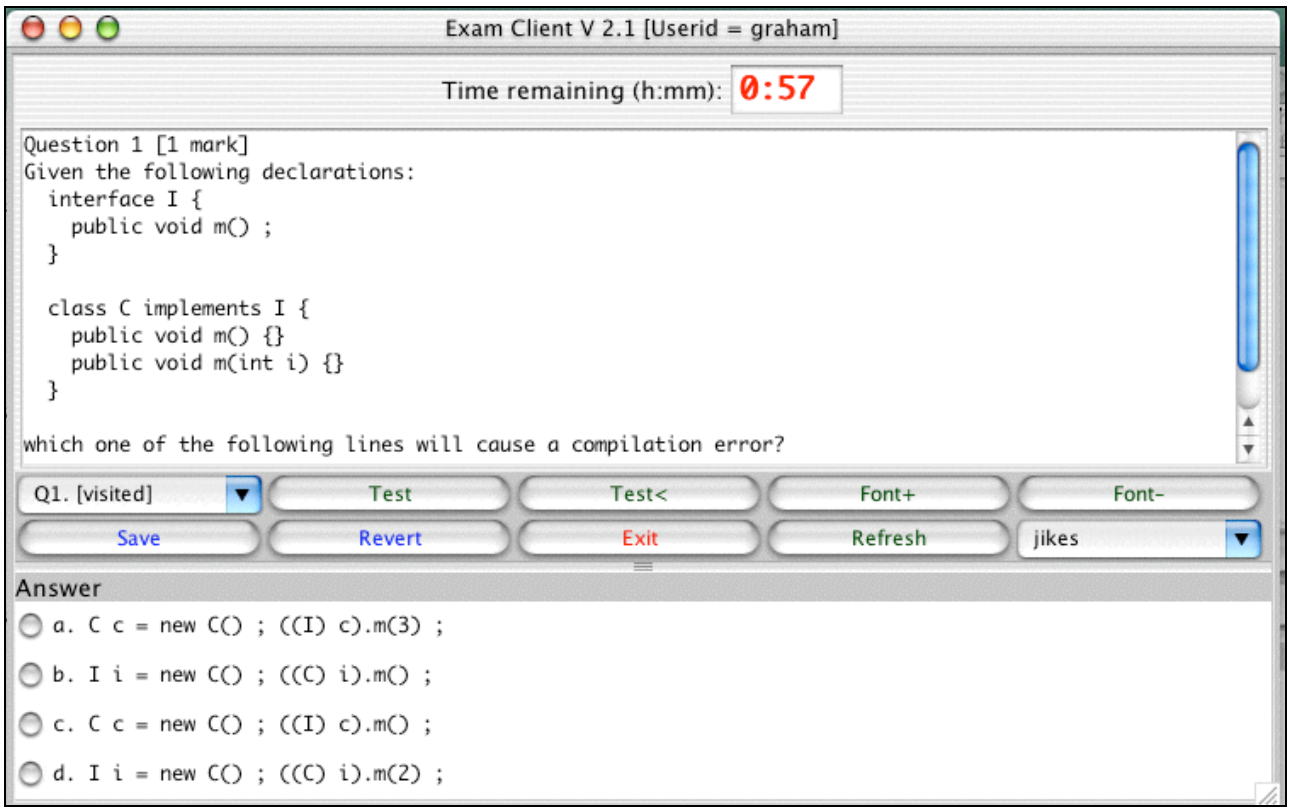

Figure 2: Initial Screen (multiple choice question)

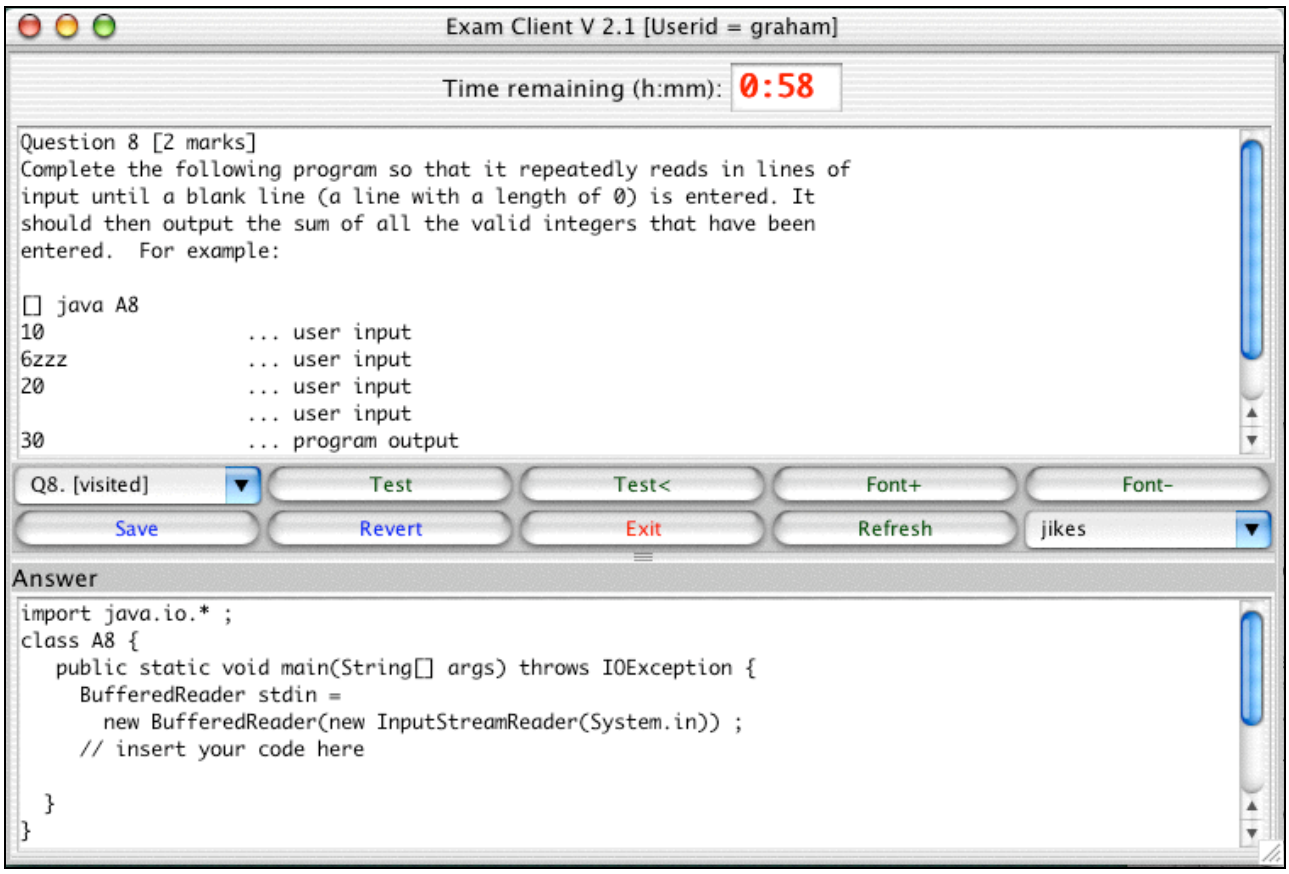

Figure 3: Programming Question

d. A variable height answer area. For multiple choice questions this consists of radio buttons and for programming questions it is a text area that initially contains skeleton code.

2. A message window that is divided into two variable sized areas (see figures 5, 6 and 7):

   a. An area in which diagnostic messages are displayed. For example, compilation error messages and runtime exception messages.
   b. An area used for data entry and the display of information related to the differences in expected program output and actual output.

The Rationale behind the division into windows and areas is based on the idea that simplicity leads to ease of use - creating too many windows makes the interpretation of the actions of the application difficult to follow, particularly in a stressful examination context. A user should know immediately where to look for anticipated output and how to control the application to maximize its effectiveness. The areas within each window allow easy (relative) resizing of areas with key roles. The initial configuration consists of the two windows, side-by-side, occupying most of the screen.

3.3 Examples

A client's session begins with entering their password (see figure 4).

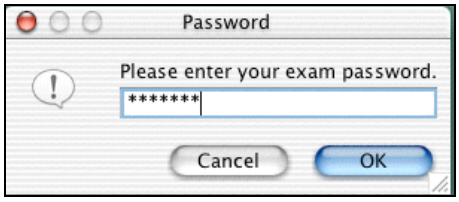

Figure 4: Login Dialogue Box

If the password is correct, a connection will be established with the server. The two primary windows will then be displayed, with question 1 being displayed in the question area in the question/control/answer window (see figure 2). Figure 3 shows a programming question being displayed together with the skeleton answer. The answer window is a Java Swing text-area component that allows program editing though its predefined editing functions. While it would be clearly better to implement a more sophisticated interface, for questions requiring only small segments of code it is less of an issue. This will be discussed further in section 6. For multiple-choice questions, each option is displayed with a radio button that allows its selection. Multiple-choice questions can either be multi-valued (one or more options can be selected) or single-valued (only one option can be selected). Figure 2 shows an example of a multiple-choice question.

For programming questions, the typical edit/compile/execute cycle has been implemented as a single operation. The point at which a student should enter code must be specified in the skeleton (figure 3 has the comment line "`//enter your code here`"). After entering the code, the *test* button is used to tell the tool to compile the answer and, if the compilation is successful, execute the program with its pre-defined test data. If compilation is not successful, the line containing the first error is highlighted and the compilation error message is displayed in the top area of the message window (see figure 5).

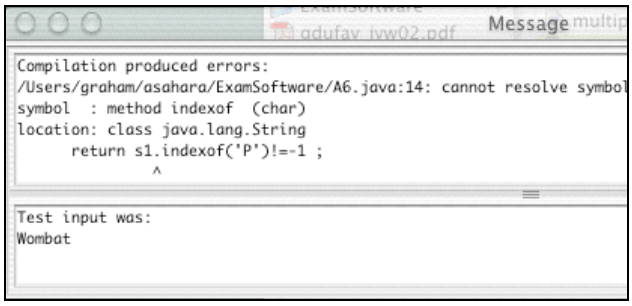

Figure 5: Syntax Error Reporting

If compilation is successful, the tool will run the program on test data and report if there were differences between the expected output and the output produced by the student's program. Figure 8 gives an example of the output produced if there are differences. The tool displays all the output before the error and the next line after it (if it exists) to ensure the context is clear to the user.

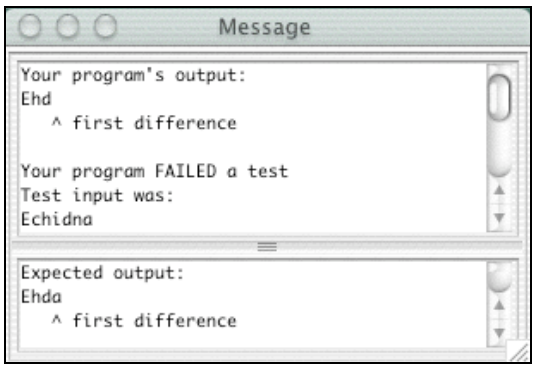

Figure 6: Output Differences

Although it is not possible to predict infinite loops, the tool takes the pragmatic approach of terminating execution after a fixed period of time. This covers the usual cases of infinite computations and the less common case of the program waiting for a non-existent event. The only potential danger is that a correct program may be terminated due to the host system being overloaded and this execution taking an inordinate amount of time. A generous period has been allowed for a program to complete its execution in the hope that this will avert miss reporting of infinite computations most of the time. In the case of computations that are judged to be non-terminating, the test input is displayed and an appropriate message displayed (see figure 7).

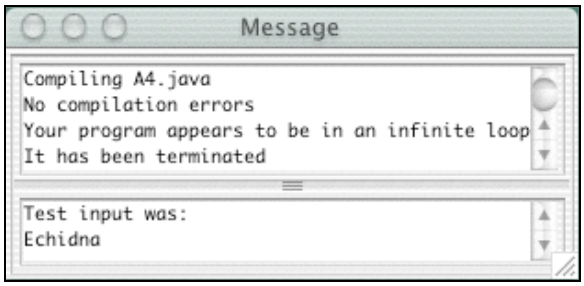

Figure 7: Infinite Loop Detection

Prior to each attempt to compile and run a program, the source code is transmitted to the server where it is saved to disk. Previous versions are lost, although it would be clearly beneficial for earlier versions to be recoverable. There are reasonable arguments that support the idea that providing a wealth of options typically increases the complexity of the tool and can have an impact on a student's ability to make effective use of the tool, particularly in the case of novice programmers.

Once a student has successfully answered a question (the status of the current question is displayed on the question menu button), or has decided to skip to another question, they can use the question menu to review the status of each question and select the next question to answer.

A question can be in one of the following states:

- *Unvisited* – the question has yet to be viewed
- *Visited* – the question has been viewed but not answered

A multiple-choice question can have the following status:

- *Answered* – at least one option has been selected

A programming question can be in one of the following states:

- *Correct* – the answer has been run and its output was the same as the expected output.
- *Error(s)* – the testing of the program resulted in a compilation error, an execution error or the resulting computation was non-terminating.

Figure 8 gives an example of the question selection menu showing various questions and their corresponding statuses.

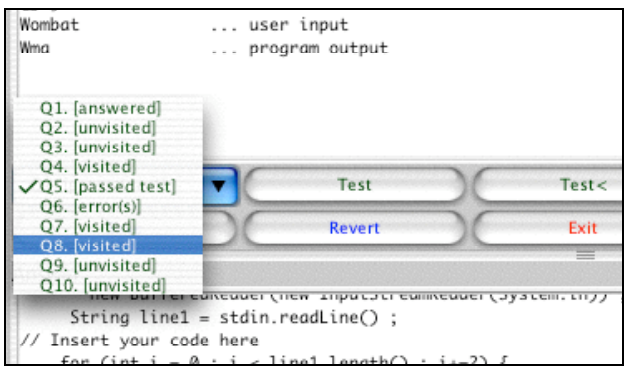

Figure 8: Question Menu

Other buttons provided in the control area are:

- *Save* – send the current answer to the server to be stored on disk.
- *Test* – test the current answer with the default test data.
- *Test<* - test the current answer with input provided by the user (in the bottom message area).
- *Revert* – revert to the original program provided for the question
- *Refresh* - Revert to the last version of the answer.
- *Font+* - Increase the size of the font used to display text.
- *Font-* Decrease the size of the font used to display text.
- *Exit* – Exit the client.

Each critical action associated with a button requires confirmation though a dialogue box.

3.4 Limitations

The client will run on any platform that has a Java Development Kit, version 1.2 or later (JDK 2002), and that can execute command scripts. The client has been tested on Solaris, Windows NT and Mac OS X. It is packaged as a JAR file and can be downloaded from the Web.

Server scripts are Unix specific and while versions of the scripts could be provided for different platforms, it would be clearly beneficial to include their functionality directly in the server.

4 Configuration and Set-up

An examination is set up by creating a directory with the following contents:

- Question files named Q1, Q2 etc. where the file specifies one of the following types of questions. Part of that specification must be the number of marks that have been allocated to the question.
    - Simple multiple-choice (only one option can be selected)
    - General multiple-choice (one or more options can be selected)
    - Programming question
- Files named Q1.in, Q2.in etc. containing input data for testing during the examination.
- A directory named "test" that contains files named Q1.in, Q2.in etc. containing input data for marking.
- A file called UIDS that must contain the usernames, passwords and class of candidate ("normal time" or several levels of "extended time") for all valid candidates.

Part of setting up an examination is to provide solutions to all questions. This can be done using a client logged in

as a supervisor. During the marking process the supervisor's answers to programming questions are run and their output stored so that it can be compared with the output of candidates' answers. The duration, and the durations of extended time examinations, must be specified when a server is started.

5 Experiences

The examination tool has been used for two consecutive years in a level two, Java programming course whose primary aim is to consolidate the programming skills and knowledge gained in the introductory programming course. The course's focus in on improving the programming proficiency and the assessment has been designed with that in mind. Students sit two online examinations, one mid-semester and the other in the usual examination period.

One of the main complaints students had with examinations that required the writing of code segments was they did not have access to the usual information and tools they had in the normal programming context (such as the Java API documentation and topic Web pages); a deficiency the tool was designed to address. While this problem was mostly solved with the examination tool, students reported the follow concerns (very vocally!) the first time the tool was used.

- The duration of the examination was too short. Although this is not an uncommon complaint for most examinations, the protests were clearly more heart felt than usual.
- The opportunity to demonstrate competence was too limited. In particular, since programming questions were graded as either incorrect (0 marks) or correct (whatever marks were allocated to the question) students felt that it was wrong to not reward solutions that were almost correct.
- The reporting of incorrect output (differences between expected and actual output) was not always clear.
- More practice with trial examinations was needed.
- The development environment did not match their usual IDE, GRASP (GRASP 2002).
- The indexing of the topic's web pages was inadequate.

While most of the concerns were easily addressed, or could be addressed with more development time and cooperation with other researchers (for example, using a grasp-like window for the answer area), by far the most pressing issue to address was that of "sufficient opportunity to demonstrate competence".

Drawing on past experience in using a simple form of an online exam in a course that taught programming in Scheme (Abelson, Sussman and Sussman 1996), we anticipated that designing a question set that ranged in difficulty from basic to difficult would be sufficient to ensure the correspondence between competence and achieved marks. While this was relatively easy to achieve with Scheme, in the case of Java, with its less regular semantics and greater complexity, particularly in is type structure and API, it was apparent that "sufficient opportunity to demonstrate competence" was not achieved to a reasonable degree.

For the final exam, despite attempting to ensure there were sufficient basic competence questions, with a gentle gradient in the difficulty of questions, the resulting raw mark distribution, a double bell curve, indicated there were still problems with marginal students being giving sufficient opportunity to demonstrate their competence. It could be argued that, at least for the 2001 version of the topic, the online examinations were only one of three types of assessment components and thus there were opportunities to demonstrate competence in the written examination, the goal was to use only online examinations.

At least some of the poor performance of marginal students could be attributed to poor examination technique, in particular, spending disproportionate amounts of time on some answers because they were "very close to working" and subsequently not having enough time to make reasonable attempts at other questions. Prior to the examination students were advised to apportion their time roughly equally between questions but clearly some students chose to ignore the advice.

Since the course included some material on graphical user interfaces, the examination tool needed to allow for questions that required the use of graphical components. Questions were designed with code that interrogated graphical components to produce textual output that could be compared in the usual way. Further, events (for example, button presses), separated by sufficient delays to allow students to see the effect on the GUI, were artificially constructed by the test code to allow answers to be fully tested. Segments of code from the skeleton answer to one of these types of questions are given in figure 9.

6 Conclusions and Future Directions

The examination tool described in this paper proved to be a self-contained, easy to use, programming environment that facilitates the development, testing and marking of programming tasks in addition to the presentation and marking of "standard" examination questions. The tool takes responsibility for many of the tasks that program development normally requires so that the student can focus on the task of writing program source code. Our experiences with its use in a Java programming topic for two consecutive years have been generally positive. Two of the key issues are the design of questions that primarily examine one aspect of programming and the construction of a question set that gently ranges in difficulty from basic to difficult, ensuring marginal students have sufficient opportunity to demonstrate their knowledge and skills.

```
class A13 {
   public static void main(String[] args) …
     MyFrame p = new
                MyFrame(stdin.readLine()) ;
     … // code omitted
     for (int i = 0 ; i < reps ; i++) {
       ((MyButton)p.getComponent(0)).
           setBackground(Color.blue) ;
        try {
          Thread.sleep(1000) ;
        }  catch (Exception e) {} ;
        … // code omitted
       ((MyButton)p.getComponent(0)).pe(
          new ActionEvent(
            p.getComponent(0),1,"action"));
     }
     System.out.println(p.getComponent(0)) ;
     System.exit(0) ;
   } // end main
}
class MyButton extends Button
            … // code omitted
class MyFrame extends Frame {
  MyFrame(String name) {
    super(name) ;
    … // code omitted
// Your code goes after this line - add a
// BListener to b1

  } // end of MyFrame constructor
}
class BListener implements ActionListener {
// More of your code goes after this line.
// Complete the declaration of the class
// Blistener

}
```

Figure 9: Skeleton Answer Code Segments

A step towards remedying one of the most fundamental problems with the tool, that is does not allow for the awarding of partial marks for approximate solutions, would be to test programs on several sets of data, ranging from simple to comprehensive. Both the in-exam testing component and post-exam marking component of the tool could easily be modified to allow testing on several data sets. The more problematical issue is the design of tasks where sub-tasks can be cleanly separated for individual testing.

The tool has some deficiencies that will be remedied in future versions. In particular, a more sophisticated, or even "pluggable", program-editing window would allow students to use a familiar editing tool. Also, to be of general use, documentation needs to be provided that would cover installation, configuration and a guide to constructing questions.

Finally, we acknowledge that we have not made reference to a number of other developments in the area that relate to the automation, to varying degrees, of the testing and grading of programming exercises (for example, REEK 1989 and KAY et. al. 1994). In developing our system further, we will draw on the experiences of the researchers involved.

7 References

ABELSON, H., SUSSMAN, G.J. and SUSSMAN, J. (1996): *Structure and Interpretation of Computer Programs – 2nd Edition*, MIT Press, Cambridge, Massachusetts.

GRASP (2002): Graphical Representations of Algorithms, Structures, and Processes, available at http://www.eng.auburn.edu/department/cse/research/grasp, accessed 23/9/2002.

JACKSON, D. and USHER, M. (1997): Grading Student Programs using ASSYST, SIGCSE 1997.

JACKSON, D (2000): A Semi-Automated Approach to Online Assessment, *Proceedings of the 5th Annual Conference on Innovation and Technology in Computer Science Education*, Helsinki, Finland.

JDK (2002): available at http://java.sun.com, accessed 23/9/2002.

JIKES (2002), available at http://oss.software.ibm.com/developerworks/opensource/jikes/, accessed 23/09/2002.

JOY, M. and Luck, M. (1998): Effective Electronic Marking for On-line Assessment, *Proceedings of the 3rd Annual Conference on Innovation and Technology in Computer Science Education*, Dublin, Ireland.

KAY, D. G., ISAACSON, P. C., SCOTT, T. A., and REEK, K. A (1994): Automated grading assistance for student programs (panel presentation). In *Proceedings of the 25th SIGCSE Technical Symposium*, p. 381.

MCCRACKEN, M. (2001): Assessment of Programming Skills of First Year CS Students: Do they know how to program. Working Group Proposal, *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, Cantebury, UK.

PRESTON, J.A. and SHACKELFORD, R. (1999): Improving On-line Assessment: an Investigation of Existing Marking Methodologies, *Proceedings of the 4th Annual Conference on Innovation and Technology in Computer Science Education*, Cracow, Poland.

REEK, K. (1989): The TRY system – or – how to avoid testing student programs. In *Proc. 20th SIGCSE Technical Symposium On Computer Science Education*, pp 112-116.

WebCT (2002): available at http://www.webct.com/, accessed 23/9/2002.

WILLIAMS, L.A. and KESSLER, R.R. (2000): Introducing Pair-Learning into Computer Science Education: *Journal on Computer Science Education*.