# GAME: A Generic Automated Marking Environment for Programming Assessment

Michael Blumenstein, Steve Green, Ann Nguyen and Vallipuram Muthukkumarasamy
*School of Information Technology, Griffith University*
*PMB 50 Gold Coast Mail Centre, QLD 9726, Australia*
*E-mail: {M.Blumenstein, A.Nguyen, V.Muthu}@griffith.edu.au*

## Abstract

*In this paper, a Generic Automated Marking Environment (GAME) is proposed for assessing student programming projects and exercises with an aim to facilitate student-centred learning. GAME has been designed to automatically assess programming assignments written in a variety of languages. The system has been implemented in Java and contains marker modules that are tailored to each specific language. A framework has been set in place to enable easy addition of new marker modules to extend the system's functionality. Currently, the system is able to mark programs written in Java and the C language. To use the system, instructors are required to provide a simple "marking schema" for any given assessment item, which includes pertinent information such as the location of files and the model solution. GAME has been tested on a number of student programming exercises and assignments providing encouraging results.*

## 1. Introduction

Automatic assessment systems have received considerable attention in the last few decades. Of particular interest is the development of tools for automatic assessment of computing programs. These advances have in part been prompted by the changing roles that are being ascribed to teachers as new learning paradigms are being adopted. A teacher is no longer simply seen as a lecturer and a supervisor, instead the role assumed is that of someone that can assist students in their ability to learn by providing an adequate learning environment, and very importantly, by providing sufficient feedback on students' work [1]. Although this is achievable in fairly small classes, this instructing methodology is not easily transferable to courses involving hundreds of students (this is especially true of introductory programming courses at the tertiary level).

Large computer science courses require students to implement hundreds of programs per semester. Hence the potential marker is faced with a difficult and time-consuming task. Automatic systems have been investigated to address this challenge to improve the consistency, accuracy and efficiency of marking assessment items [2]. In addition such systems can assist in providing timely feedback to students over a large number of assessment items [1]. Finally, teachers have found that such tools enable them to perform more efficiently by concentrating human resources on tasks that cannot be automated. However, although some systems already exist there is still much research that must be undertaken in order to develop an accurate marker that can be used for small and large computer programs and can be accurately used over a range of programming languages and assessment items.

The remainder of this paper is divided into five sections. Section 2 reviews the systems currently in use for automatic assessment, Section 3 explains the context of GAME and its functionality, Section 4 presents some experiments conducted using GAME, whereas a discussion of results and experiences with the tool takes place in Section 5. Finally, conclusions and future work are described in Section 6.

## 2. Current systems

In an effort to address the challenge of marking large volumes of electronic assessment, a number of automated systems have been developed and tested [2]-[6]. In a recent study it has been asserted that automatic marking of computing assessment provides advantages not only to the teacher, but may also play an important role in student learning outcomes [1]. Reek [3] details a system called TRY for grading students' PASCAL computer programs by comparing the outputs of their program against a "model" output. Their system does not take into account style or design issues. Jackson and Usher [2] proposed a system called ASSYST that checks the correctness of an ADA program by analysing its output and comparing it to a correct specification using in-built tools of the UNIX operating system. Saikkonen *et al* [5] proposed a system

Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)
0-7695-2108-8/04 $ 20.00 © 2004 IEEE

called Scheme-robo for automatic assessment of "Scheme" programs by analysing the return values of procedures and the structure of student code. The return value could easily be compared to a model solution whereas the program code was reduced to a list structure that could be analysed to determine whether it contained particular sub-patterns specified by the instructor. Finally Ghosh *et al* [6] developed a preliminary system for computer-program marking assistance and plagiarism detection (C-Marker). It used a syntax dependent approach for C assignments. The marking component simply compiles and executes each student program and performs a simple comparison between the program's output and a model output file. The system's plagiarism detection module is tightly integrated with the marker and is based on the examination and comparison of each program's structure using a physical and logical profile.

### 2.1 Drawbacks of current systems

The main drawback of each of the above-mentioned systems was that they were tested on one particular programming language. Although it was mentioned that some of the systems could be extended to operate on programs of other programming languages, an investigation was not conducted to determine the feasibility. Another deficiency inherent in some systems is their inability to deal with a wide variety of assessment items. In some cases the criteria for marking the correctness of student programs is hard-coded and requires the system to be updated regularly. This is a substantial limitation of these systems and needs to be addressed.

## 3. The proposed tool

GAME has emerged building on a previous system for marking C assignments [6] (C-Marker) and was designed to address the limitations of the C-Marker system. An overview of GAME is presented in Figure 1. Currently, the plagiarism module (adapted from C-Marker) is inactive, however in future it will play an integral part in the overall system. GAME will be made accessible to students for marking/verifying their own assignments.

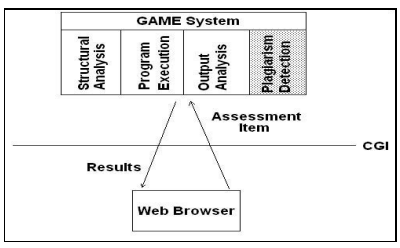


**Figure 1 Overview of GAME**

### 3.1 Overview of C-Marker

The C-Marker system was originally developed to mark simple command-line C code assignments on a Unix platform. C-Marker separated the marking of C assignments into two components: 1) Testing the C code structure, and 2) examining the correctness of the student program output.

The approach used by C-Marker to examine a program's structure and correctness was very static, as it did not easily allow more than one type of assignment to be marked. Also, C-Marker could only compile and execute a single C source file under Unix. In addition, the criteria for marking students' programs were hard coded into the C-Marker system. These short comings needed to be addressed, and a more dynamic approach was required for marking different types of assignments, and applying different marking strategies for looking at source code structure and testing the correctness of the program's output.

The GAME system developed in SDK 1.4.1 has built upon the C-Marker system. The main focus of the GAME project was to develop an automated system that could dynamically mark different types of programming assignments, mark assignments written in different programming languages, apply a generic strategy for looking at source code structure, and apply different types of marking strategies for examining the correctness of a student's program output.

### 3.2 The marking schema and marking strategies

To be able to dynamically mark different types of programming assignments, The GAME system requires the assessor to fill out a marking schema. The marking schema contains information about assignment files, assignment marks, and marking criteria.

A student's program may require different files to execute. These files fall into four main categories: input, output, correct output, and instruction files. The *input file* is only necessary if the student's program requires data to populate its data structures, the *correct output file* contains the correct output expected from the student's program, the *output file* stores the output from the student's program, and the *instruction file* holds the instructions necessary for the program's operation. If the file is not required then a "null" is used, otherwise a file can be read or written to (standard input or standard output respectively).

The marking criteria can be specified for different types of assessment. The marking criteria allow the assessor to apply different marking strategies to test the correct result from students' programs. The ability to apply different marking strategies to a variety of

programming assessment is an important part of building a generic marking system and there is no single type of marking strategy for all types of assessment.

At present there are two marking strategies: a "keyword" strategy that examines the student's program output for a keyword at any position and an "ordered-keyword" strategy, that looks for an ordered set of keywords in the student's program output. Both strategies attempt to ignore irrelevant information by skipping those words that do not match the keyword (i.e. correct output).

The creation of a marking schema for an individual piece of assessment provides a convenient way for a course convener to enable other members of a teaching team to mark their programming assessment. The schema simplifies the use of the GAME system, as there is no need to deal with low-level details such as the method of input to the student's program. The marking schema will play an important role in future versions of GAME, and eventually it will enable teachers to build marking schemas without understanding the schema's structure.

### 3.3 Dealing with different languages

One of the aims of the GAME system was to enable the marking of programming assignments written in different languages. At present GAME can mark Java and C assignments. This is possible as both Java and C languages are compiled and then executed. To enable a generic base class to compile and execute any programming source code, the common behaviors need to be recognised (see Figure 2), so that the sub-classes required to mark different programming languages can inherit a common behavior and provide their own concrete implementation for their type of programming language. This provides a simple framework for adding additional languages to GAME in the future (if required).

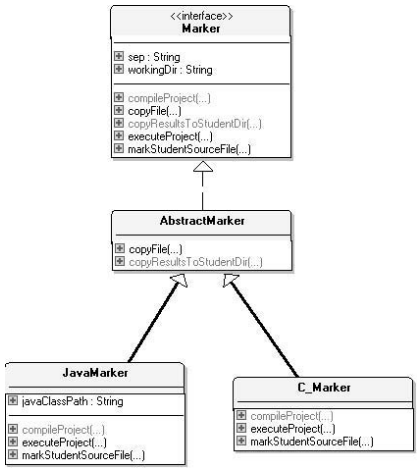


**Figure 2 Class diagram for compilation and execution of different programming languages**

A requirement for the execution of the GAME system is that the different programming environments are setup on the computer that the system is running on. At present, GAME has only been tested under Windows, but should work with some minor modification on different platforms.

### 3.4 The GUI interface

The GAME GUI interface provides a straightforward interface to enable the user to mark different types of programming assessment (see Figure 3). The user is first required to select the root directory containing all student folders with their programming assessment.

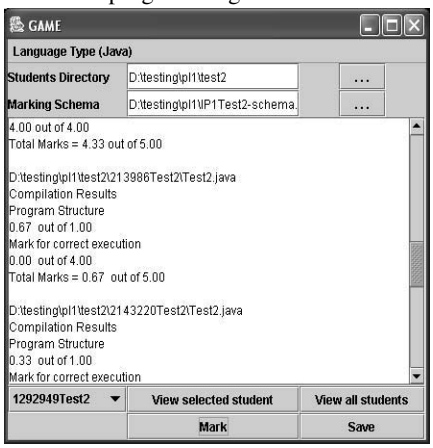


**Figure 3 GAME user interface**

The second requirement is the marking schema for the type of programming assessment being examined (see Section 3.2).

Once the above parameters have been entered, the user may select the programming language from the "Language Type" menu option (see Figure 3), and may then press the "Mark" button. When the students' programs have been marked, a summary of the students' results is displayed. A detailed report of a particular assessment piece may be obtained by selecting the desired student's directory from the dropdown combo-box and then pressing the "View Selected Student" button.

### 3.5 Reporting information

As mentioned above, the results are displayed in a summary format that allows the marker to browse the overall results for each student (see Figure 3). This summary format shows a mark for source code structure, a mark for correctness of the student's program output, and a list of warnings or compile-time errors (if any).

The mark for structure is calculated in three parts: 1) the number of block comments compared to the number of

functions, 2) the valid declaration of variables (both global and local), and 3) the number of "magic numbers" that are used in the code. Each of the three parts is worth 1/3 of the overall "structure" mark.

The second mark calculated is for the correctness of the student's program output. Based on the strategy used to mark the correctness of the student's program output, a percentage of the mark is displayed. A breakdown of results for a particular student may be seen in Figure 4.

```
D:\testing\pl1\test1\343434\Test1.java
Compilation Results
Program Structure
1.00 out of 1.00
Number of block comments = 2
Number of functions = 1
Number of single variables = 0
Number of magic numbers = 1
Mark for correct execution
* * Program did not compile * *
D:\testing\pl1\test1\343434\Test1.java:30: ';'
expected for (int i = 1; i < exponent); {
^
...
4 errors
Total Marks = 1.00 out of 5.00
Output from program
* * No output file found * *
```

**Figure 4 Results for a student's assignment**

## 4. Experiments

To test the capabilities of GAME, a number of experiments were conducted using real-world data. Two different types of programming assessments were marked and the results obtained from GAME were compared to those obtained from a human marker.

### 4.1 Java programming exercises

The first experiment utilised a set of in-class lab assessments obtained from the first year programming course at Griffith University. Students were required to write a small Java program that obtained some input from the user, and output some result. The test came in three varieties to prevent plagiarism.

Two of the three test types were used in the experiments, which resulted in a total of fifty assessment items presented to GAME. As the assessment was fairly simple, program structure was not taken into account for the marking process. However, a mark was allocated for the correctness of program output. GAME's performance compared well to the human marker. In forty-one out of the fifty assessments (82%), GAME provided an identical mark to its human counterpart.

### 4.2 C programming assignments

The second assessment type was a major C programming assignment for a Masters programming course. In this assignment, the students were required to read in a data file to populate their programs' data structures. Next, based on different command-line menu options, their program was required to output the result based on the selected command. These programs were more difficult to mark as their output contained more complex information, including irrelevant formatting information such as borders, underlines etc. The unnecessary information output by some students' programs highlighted the difficulty of marking assessment using an automated system. A human marker is not faced with such problems and may examine a student's result concentrating only on information that is useful.

Twenty of the above assignments were marked by GAME and there results were evaluated. The assignments were not written for automated marking and presented some challenges. The first challenge was the way in which a user was required to select a menu option and provide the necessary input for that particular command. To mark these programs an instruction file was created that held the menu commands and input data (see Figure 5). The instruction file was then redirected to the individual program's standard input.

```
1 2 3 4 9687 5
```

**Figure 5 Instruction file for C programming assessments**

The instruction file in Figure 5 holds the menu commands for options 1, 2, 3, 4, and 5. The menu 4 command required a second input "9678". The programs that abided by the programming instruction ran correctly under the GAME system, however some programs were written with an alternate menu structure, or left out certain menu commands. These programs could not be marked by GAME, and were not included in the experiments.

The results for the twenty assignments were fifteen out of the twenty showed very similar or identical results to the human marker (75%).

## 5. Discussion

### 5.1 Java programming exercises

Of the nine exercises (out of fifty) where GAME did not agree with the human marker, some interesting observations were made. For six exercises the human marker awarded part marks, whereas GAME gave zero. This disparity may be explained by the fact that the human marker manually looked through the code and gave some marks if the basic algorithm was present or if the program actually executed but did not give any result. In such cases, a small amount of human intervention may be necessary. In future, GAME may be enhanced so that any programs obtaining a zero may be flagged for further

scrutiny by the human marker. Also, GAME may be modified to provide part marks if the program compiles and executes.

In one of the exercises, the human marker gave part marks (when GAME gave full marks) because the program's output was correct, however it contained additional output that was not requested. GAME gave full marks because it found the correct answer based on the keyword marking strategy. In this case, it ignored all supplementary information. This situation may be avoided in the future by simply using a different strategy, or adding a new strategy to the framework for coping with specific situations.

Finally, for the last two exercises where GAME disagreed with the human marker, it was found that the error could be attributed to the human! In this case, the human allocated full marks whilst GAME calculated zero. An instance such as this highlights the usefulness of GAME as an aid for crosschecking the marks assigned by teachers.

### 5.2 C programming assignments

In five cases out of twenty, the marks given by GAME either did not correlate well with the human marker or the program could not be executed. Out of the five problem cases, one assignment did not execute due to the fact that the student's program attempted to read an incorrect input file name. Irrespective of how carefully the assignment requirements are outlined, there will still be those students who don't follow the assignment guidelines and name their input and output files incorrectly. One way to overcome this problem for command-line programs is to instruct students to write their results to standard output, which will enable the information to be redirected to a constant "results" file name. Also, instructions can be given to students' programs via standard input, again avoiding complications with file naming conventions. The GAME system enables the use of redirection very simply and may be a feasible way of dealing with the above problem.

The remaining four assignments showed very different results to the human marker. The main reason for this disparity was that the output files generated by the student programs were tightly coupled with the formatting text (as explained earlier). At this stage, automating marking for this type of assignment would require that the output for the different commands be separated from the formatting information.

## 6. Conclusions and future work

This paper presented an automated system called GAME for marking programming exercises and assignments in university programming courses. The generic nature of the system makes it a powerful tool for marking different assessment items with the use of a simple "marking schema", in addition to marking assessments written in different languages. Currently, programs written in C and Java may be marked by the system, and the system's architecture allows for simple extension to other languages. The results obtained comparing GAME's performance to a human marker are very encouraging.

In future, further marking strategies will be added to the system to deal with issues relating to the variety of output obtained from students' programs. Also, a more detailed examination of the source code structure will be explored, including the flow of execution. In addition, the plagiarism detection module will be investigated in the context of the overall system. Finally, in coming semesters, a version of GAME will be made available to students so that they may compile, execute, obtain feedback and fine-tune their assignments prior to the submission deadline, effectively facilitating student-centred learning.

## Acknowledgments

This work was supported by Griffith University as part of New Researcher and Teaching Grant Schemes.

## References

[1] L. Malmi, R. Saikkonen, and A. Korhonen, "Experiences in Automatic Assessment on Mass Courses and Issues for Designing Virtual Courses", *Proceedings of The 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE' 02)*, Aarhus Denmark, (2002), pp. 55-59.

[2] D. Jackson and M. Usher, "Grading student programs using ASSYST", *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education, San Jose, California, USA*, (1997), pp. 335-339.

[3] K. A Reek, "The TRY system or how to avoid testing student programs", *Proceedings of SIGCSE 1989*, (1989), pp. 112-116.

[4] J. English, and P. Siviter, "Experience with an automatically assessed course", *Proceedings of The 5th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'00)*, Helsinki Finland, (2000), pp. 168-171.

[5] R. Saikkonen, L. Malmi, and A. Korhonen, "Fully automatic assessment of programming exercises", *Proceedings of The 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE' 01)*, Canterbury, United Kingdom, (2001), pp. 133-136.

[6] M. Ghosh, B. Verma, and A. Nguyen, "An Automatic Assessment Marking and Plagiarism Detection", *First International Conference on Information Technology and Applications (ICITA 2002)*, Bathurst, Australia, (2002).