

Performance Analysis of GAME: A Generic Automated Marking

Environment

Michael Blumenstein¹, Steve Green¹, Shoshana Fogelman², Ann Nguyen¹ and Vallipuram Muthukkumarasamy¹

¹*School of Information and Communication Technology, Griffith University, PMB 50 Gold Coast Mail Centre Bundall, QLD 9726 Australia*

²*School of Environmental and Applied Sciences, Griffith University, PMB 50 Gold Coast Mail Centre Bundall, QLD 9726 Australia*

Abstract

This paper describes the Generic Automated Marking Environment (GAME) and provides a detailed analysis of its performance in assessing student programming projects and exercises. GAME has been designed to automatically assess programming assignments written in a variety of languages based on the "structure" of the source code and the correctness of the program's output. Currently, the system is able to mark programs written in Java, C++ and the C language. To use the system, instructors are required to provide a simple "marking schema" for each given assessment item, which includes pertinent information such as the location of files and the model solution. In this research, GAME has been tested on a number of student programming exercises and assignments and its performance has been compared against that of a human marker. An in-depth statistical analysis of the comparison is presented, providing encouraging results and directions for employing GAME as a tool for teaching and learning.

Keywords: architectures for educational technology system; post-secondary education; programming and programming languages; teaching/learning strategies

1. Introduction

Over the past few years, the development of tools for automatic assessment of computing programs has generated considerable interest. The desire to advance these learning tools has been in part prompted by the changing roles that are being ascribed to educators, as new learning paradigms are being adopted. An educator is no longer seen as a lecturer or 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 sufficient student feedback has been achievable in fairly small classes, this instruction methodology has been more difficult to apply to courses involving hundred of students, as well as courses undertaken through long distance education. Hence, automatic systems are being investigated not only to address student feedback issues, but also to improve the consistency, accuracy and efficiency of marking assessment items in large computer science courses [2]. This could allow educators to perform more efficiently by concentrating on tasks that cannot be automated, such as helping students grasp larger concepts and ideas.

Current automated marking systems are advancing in their ability to accurately mark students' programming assessments, but there is still much research that must be undertaken 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. Specifically, a single generic marking system is less costly to maintain and is more convenient to use than a number of smaller ones.

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]-[8]. 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. 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. Saikkonen *et al* [5] proposed a system called "Scheme-robo" for automatic assessment of scheme programs by analysing the return values of procedures and the structure of student code. English [7] has recently proposed a system for automated assessment of GUI-based Java programs using the JEWL library. Finally, Ghosh *et al* [8] developed a preliminary system for computer-program marking assistance and plagiarism detection (C-Marker). The system compiles and executes each student program and performs a simple comparison between the program's output and a model output file.

The main drawback of each of the systems mentioned above is that they were tested on one particular programming language. Although it was stated 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. Finally, many of the existing systems have been tested on small programs, whereas the challenge of marking larger pieces of software

has been essentially overlooked. In order to try and overcome the problems faced with current automated marking systems, GAME [9] was developed in SDK 1.4.1 and has emerged building on a previous system for marking C assignments [8] (C-Marker). It was designed to address the limitations of the C-Marker system and other existing systems. An overview of GAME is presented in Figure 1; it consists of four modules: structural analysis, program execution, output analysis, and plagiarism detection. The assignment is passed through a user interface to the GAME system and the results of marking are received as output through this interface. The first three modules have been developed and tested; the plagiarism module (adapted from C-Marker) is currently inactive, but will be further developed in future versions of the system.

The long-term goal for GAME is to make it accessible to students for marking/verifying their own assignments, as a student-centered tool for real-time feedback, as it is sometimes very difficult to obtain immediate feedback from educators. Its plagiarism detection unit should also make it easier for markers to identify similarities in assignments.

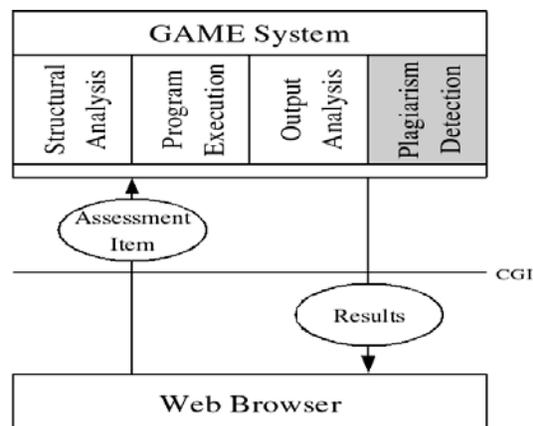


Figure 1 Overview of GAME

Based on the deficiencies of the early C-Marker system mentioned above, it was clear that the GAME system needed to address some major requirements. These may be summarized as: 1) The handling of different programming languages, 2) the handling of different assessment types, 3) the ability to implement different marking strategies to test the correct output of a variety of assessment items, 4) the handling of multiple source files and 5) the ability to accurately examine the structure of students' source code. In an earlier version, GAME addressed the first three of the five requirements mentioned above [9]. However, in order to make GAME a more robust marking system, two additional components were implemented: firstly, the ability of GAME to handle multiple source files, and secondly, the ability to accurately examine the structure of students' source code (a preliminary discussion of these may be found in [10]). This has been achieved by assessing three different aspects of source code structure and presentation in students' assignments. The source code structure is examined in terms of the number of comments in student code as compared to the number of lines of source code. GAME also performs an analysis of indentation practices and its consistency within the source code. Lastly, the programming style is marked by examining the students' approach to designing well-structured, reusable software. In each of the three sections, the student is awarded a percentage of the overall mark for each part. In its present form, GAME currently addresses all the above points in part or in full.

The following figure (Figure 2) provides a hierarchical representation of the GAME system presenting its composition and operation. The use of an adjustable marking scheme facilitates a weighted performance measure of a student's assignment based on each of the processes outlined in Figure 2. This information is summed and stored,

providing an overall result for each assessment item. GAME's performance has been evaluated on a large set of diverse programming assignments, and the results have been analysed in detail.

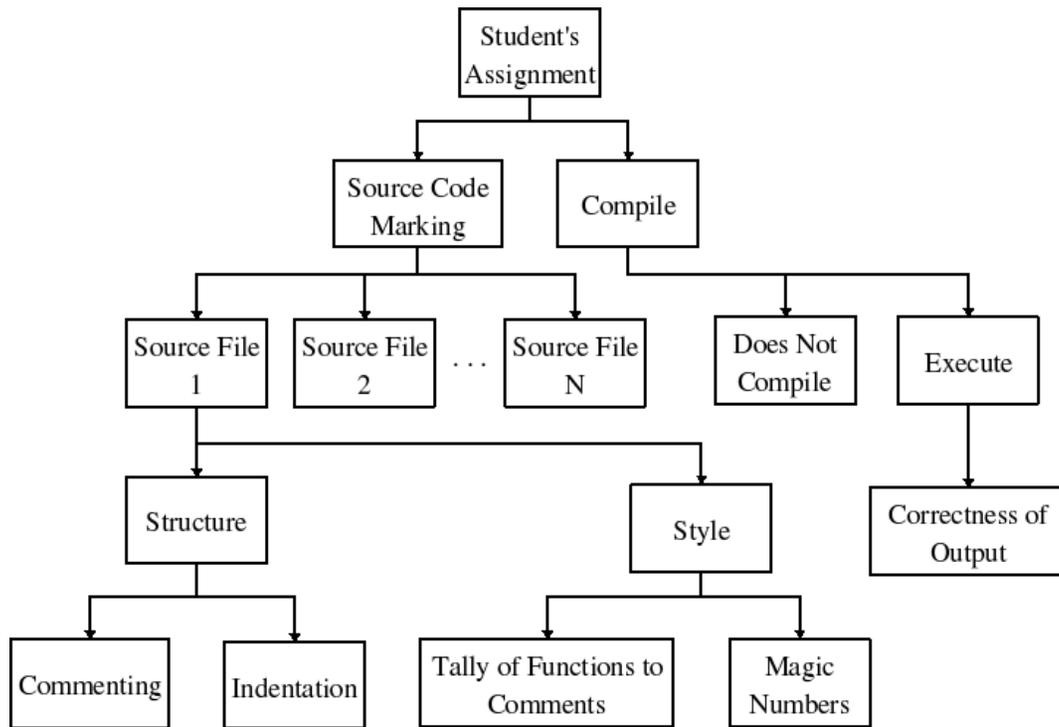


Figure 2 Hierarchical representation of the GAME system

2. Detailed Overview of GAME

2.1 Indentation

The analysis of indentation practices and their consistency within student source code is an important part of any programming assessment. Students need to understand the discipline of producing code that is maintainable and easy to read, both for themselves and other programmers. Indenting source code to an adopted standard provides the marker with a measure to examine a students' source file for correctness, and most importantly consistency of indentation standards. At present, GAME assesses source

code indentation by examining strategic "key" points in the code. These points are at the beginning of functions, control statements, and global variables. At each key point where the source code is examined for correct indentation, a counter is incremented to record the total number of successful indentations (refer to Equation 1). Hence, indentation accuracy is determined by calculating the ratio of the successful indentations i.e. those that have the correct indentation size, to the number of tests performed at key points. With this method, the GAME system only examines a percentage of the source code rather than every line. To assign indentation marks, the result obtained from GAME's analysis is compared to a set of indentation measures stipulated in the GAME schema file.

The indentation mark and size are specified in the marking schema (see Figure 3). In the example shown in Figure 3 it may be seen that the correct mark is worth three (3) and the correct indentation size is four white spaces. When examining a source file's indentation, the GAME system keeps a tally of the number of white spaces that should appear at any specific point in the source code. If a tab space is encountered, it is converted to the white space equivalent i.e. if the indentation size is four (4) a tab will be worth four (4) white spaces.

@ indentation mark is made up of two parts
@ 1) the mark allocated for the source code indentation
@ 2) correct indentation size (number of white spaces)
#indentation_mark
3, 4

Figure 3 Illustrates a section of the marking scheme that specifies the indentation mark and indentation size.

Once all points have been examined, the average correct indentation is calculated.

$$IndentAccuracy\ p = \left(\frac{NumOfSuccesses}{NumOfTests} \right) \times 100 \quad (1)$$

The above procedure is carried out for each source file contained in a student's assessment and is summed to form an overall correct indentation percentage (as defined in Equation 2).

$$c = \frac{1}{n} \sum_{s=1}^n p_s \quad (2)$$

The final mark is then calculated by multiplying the correct indentation mark by the correct indentation percentage.

2.2 Commenting

Commenting standards are again a very important aspect of writing maintainable code. The methodology by which the amount and standard of commenting may be evaluated is a very fuzzy area; there are two main points to consider: 1) how many comments should be incorporated into the source code to aid in its understanding, and 2) what constitutes a good/useful comment. Currently, for the first point, GAME examines the number of comments in a student's source code. For most common programming languages (some versions of C, C++, and Java) comments can be inline (the comment is on one single line) or they may be block comments, which span multiple lines. Again some of these areas are fuzzy, but it can be said that in most programming languages, a block comment is normally used for source file header information, function/method definitions, and anything else that requires a lengthy description. Inline comments are usually placed amongst code fragments and provide some insight into ambiguous lines of code. The second point has not been addressed in the GAME system, but is perceived to be an important step in future versions of GAME.

To calculate the commenting mark for a student's assessment, the student source files are parsed to determine four values, 1) the number of lines of code (LOC), 2) the number of inline comments, 3) the number of block comments, and 4) the number of functions/methods. Once these values have been calculated, different metrics can be used to determine, or provide some measure of each source file's commenting style. The LOC were calculated by counting the command lines in each source file, as opposed to counting each line. Counting the command lines in each source file provided a more accurate result when determining the source file commenting percentage.

One software metric, which relates to commenting for an entire source file, is the ratio of the number of comments (both inline and block) to LOC (see Equation 3). It was found through empirical analysis that a commenting percentage of 30% or greater provided a good indication of commenting in comparison to the number of LOC.

$$CommentingPercentage = \left(\frac{NumOfComments}{LOC} \right) \times 100 \quad (3)$$

A second metric, relating specifically to commenting for functions/methods, compares the number of block comments to the number of functions/methods in the source code. Good programming style would require that the student provides some description of each function in their source file. At present GAME does not verify that there is a relationship between a block comment and a function, simply the quantity of each. Marks for programming style are deducted if there are less block comments than functions. The area of evaluating programming style and structure is ongoing in the GAME system.

2.3 *Programming Style*

The programming style component of the source code structure mark concentrates on students' design, code reusability and maintainability. At present, the GAME system performs two operations with respect to programming style: 1) a comparison of the number of functions with the number of block comments (as described previously), and 2) a tally of the number of "magic" numbers (i.e. values hard-coded with no apparent meaning). Although GAME's functionality with respect to programming style is currently simplistic, it will be possible to extend this component with relative ease in future versions of the system.

2.4 *Marking Schemas and Strategies in GAME*

To be able to dynamically mark different types of programming assignments, the GAME system requires the assessor to complete a marking schema (see Figure 4). The marking schema at present is stored in a basic text format and will eventually be stored in an XML format, to ensure the correctness of parameters used for the marking schema. The marking schema currently contains information about assignment files, assignment marks, and marking criteria. The creation of a marking schema for an individual piece of assessment 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 (i.e. from standard input or from a file).

```
@ This is a schema for marking IPL lab assessment for Test1
@ Values must be comer delimited
@ Language type supported at present C, Cpp, and java
#language_type
java
#path
\projects\marker\testing\p11
#marking_Criteria
keyword
#coutput_mark
5
#compiles_mark
5
@ Structure mark for now just looks at the number of function and
magic numbers
#programming_style_mark
3
@ Indentation mark is made up of two parts
@ 1) the mark allocated for the source code indentation
@ 2) correct indentation size (number of white spaces)
#indentation_mark
3, 4
@ Commenting mark is made up of two parts
@ 1) the mark allocated for source code commenting
@ 2) the percentage level the is considered good commenting
@ This is calculated by (number of comments)/(lines of code)
#commenting_mark
4, 30
#input
stdin
inputTest1.txt
#output
stdout
results.txt
#coutput
value
81
#instructions
```

Figure 4 GAME marking schema

Currently, GAME allows the assessor to apply different marking strategies to test the result from a student’s program for correctness. 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).

2.5 Marking Multiple Programming Languages

Designing software that can only evaluate one programming language does not provide the user/marker with consistency between different programming courses. It is clear that regardless of what programming language is being taught, there is a common discipline that is required to be learnt. Students learning programming must strive to learn how to write robust and reusable code that can be understood by different programmers. To do this, the student must learn good coding design practices and standards that are expected in private industry. GAME at present is capable of marking three programming languages. Developing the implementations for other programming languages is readily achievable by extending the current model (see Figure 5) and will make GAME more robust, as a generic marking system.

From Figure 5 it may be seen that the base class “Marker” holds all the common behaviour that is required from the sub-class. In all cases, student source files are compiled, executed, and the source file examined for structure and programming style.

Each sub-class performs an inherited function in a slightly different way (polymorphic) i.e. to execute a student's program written in C a call to the ‘C’ compiler is required, whilst a Java program calls the “javac” command to compile a Java program. If a programming language does not have a compile stage i.e. Perl, an empty implementation would then be required. The empty implementation would be called but would not perform a task. All implementations of the Marker sub-class are Marker objects and therefore can be passed to a method by the base class. This means that the algorithm for marking a student's source code is unaware of a particular implementation,

and simply calls the common behaviour in the Marker class. Java at run-time dynamically binds the correct object implementation to the method call.

The concept of designing a class that describes common behaviour for all sub-classes is the essence of designing a generic system to dynamically handle different conditions. When parsing a student's source file, again when examining different languages, the same process is required, in that different aspects of the student source code are examined, and this is done in different ways for different language types. Figure 6 illustrates the base class *SourceCodeMarker* that holds all the common behaviour that is required for parsing a student's source file for structure and programming style. All three sub-classes below are the required implementations for parsing a C, C++, or java source file. If GAME was extended to parse a Pascal source file, a new sub-class of *SourceCodeMarker* could be designed to handle Pascal source files.

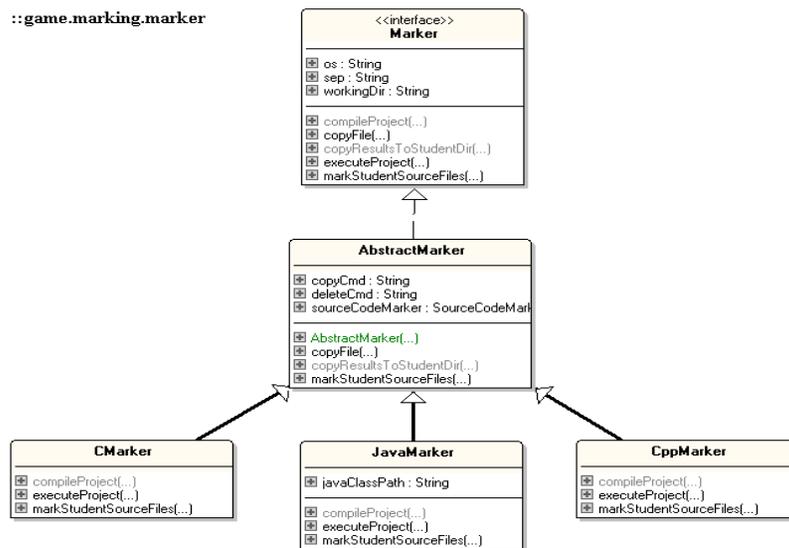


Figure 5 Illustrates a class diagram of the GAME marker class with a sub-class for marking C, C++, and Java programs.

::game.marking.parser

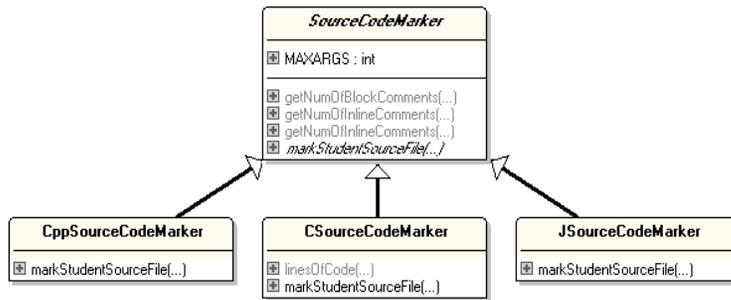


Figure 6 Class diagram illustrating the design for parsing a student source file for structure and style.

2.6 Multiple Source Files

Most programming assessments encourage their students to design their programs in a structured manner. This often requires students to develop more than one source file, in fact, most assessors would deduct marks from an assessment item if the entire student's code was stored in a single source file. Whether designing a program in a procedural or object-oriented (OO) language, there is still a need to encourage students to structure their programs, and design meaningful source files. GAME is equipped to mark assessment items that have more than one source file. All the students' source files are examined, and the processed information for each source file is stored in a list in the Student object that represents a particular student assessment (see Figure 7).

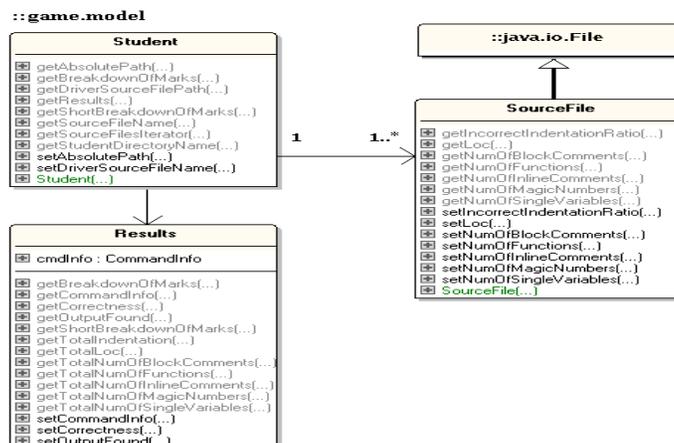


Figure 7 Class diagram showing that a Student object can store multiple source files.

As each student's source file is examined, it is inspected in two ways, 1) source file structure and 2) programming style. Each of these two components requires information to be stored about each student's source file. When each of the student's source files has been processed, the total mark for the student's source file structure and programming style can be calculated. The source files' structure and programming style is calculated by summing each of the source files' marks and calculating an average mark for each section. In each section, structure and programming style, the marks are given as a percentage of the mark allocated in the marking schema. The results for each student are generated from the Results object (see Figure 6) stored in each Student object. The Results object combines source file information along with compilation and execution information to produce the student's overall result for an assessment item. This information for each student can be generated in a brief outline of all results (see Figures 8 and 9), or a full breakdown for a particular student's result.

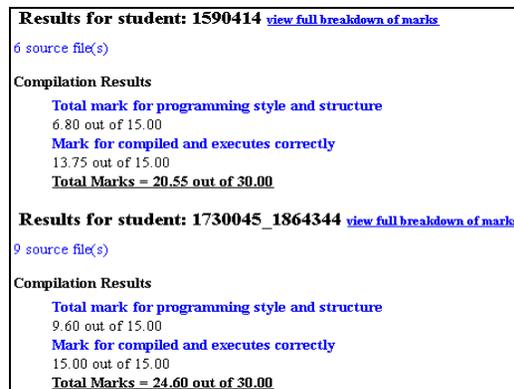


Figure 8 Displays a brief breakdown of all the students' results

2.7 The GAME GUI

The GAME GUI provides a straightforward interface to enable the user to mark different types of programming assessment (see Figure 9). In order to operate the system, the user is first required to select the root directory containing all student folders with their

programming assessment. The second requirement is the selection of an appropriate marking schema for the type of programming assessment being examined (see Section 2.4).

Once the above parameters have been entered, the user may then press the "Mark" button. When the students' programs have been marked, a summary of the students' results is displayed as a hypertext document within the system's main window. The summary includes a hypertext link to provide a detailed report of a particular assessment item.

In this study it was decided to investigate the capabilities and robustness of GAME on a number of real-world assessments to determine GAME's ability to mark different programming assignments and to evaluate the effectiveness of handling different programming languages, assessment items and multiple source files, as well as having the ability to use different marking strategies and the capacity to accurately examine the structure of source code. Its proficiency to mark is assessed by comparing results obtained with GAME to those obtained with a human marker

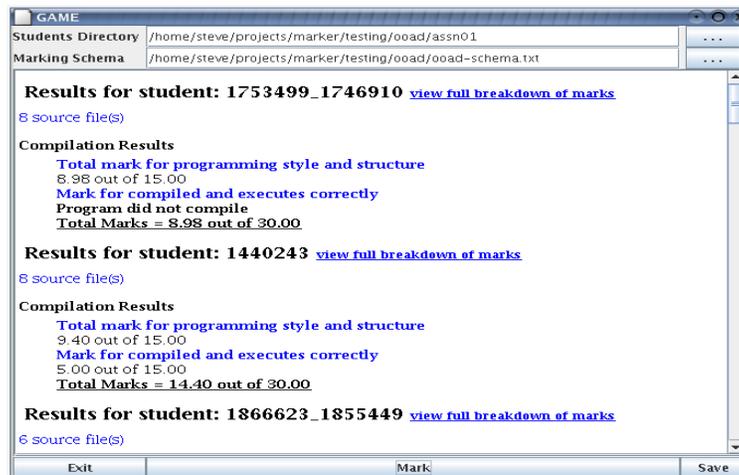


Figure 9 GAME user interface

3. Methods

To evaluate the capabilities, practical applicability and robustness of GAME in its present form, the system was used to mark a number of real-world programming assessment items. These involved different programming languages, different assessment items, multiple source files and different marking strategies. These experiments also focused attention on the structure of students' source code. The following section describes the methods employed to assess the assignment items and to evaluate the effectiveness of GAME.

3.1 *Assessment Item Collection*

GAME was evaluated on three different types of programming assessments, which were all command-line programs, two of the assessments were written in Java and one was written in C++. Java Test 1 was an in-class lab assessment obtained from a first year programming course. This assessment item involved only a small amount of coding, requiring a single source file. It was given to 3 different student groups, with a total of 98 assessment items being collected.

Java Test 2 was set in a more advanced OO analysis and design course. This assessment item was a "hand-in" assignment and involved a more complex programming structure requiring multiple (on average 7 to 8) source files. Students worked in groups of two or three. In total 23 assessment items were collected for Java Test 2.

The third assessment item used to evaluate GAME involved assignments from a first year C++ course. C++ Test 3 involved the students being provided with a partially

completed source file to base their design on. This skeleton included a command-line menu and header files. When examining the structure component of the students' source code, the GAME system only examined the C++ files written by the student and did not consider the header files. In total, 13 assignments were collected and given to the GAME system for marking.

3.2 *Marking of Assessment Items*

Each assessment item was fed to the GAME system and marked in the following manner. Java Test 1, 2 and C++ Test 3 each were given three major marks, one for structure, the second for compilation and the third for execution. The structure mark was consistent for all three tests and was derived from three categories, commenting, indentation and style with a maximum score of 5 marks being awarded to each category for successful completion. For Java Test 1 the compile and execution marks were combined and given a total of 10 marks upon successful operation. Java Test 2 and C++ Test 3 were given a maximum mark of 5 for successful compilation and a maximum of 10 marks for successful execution.

3.3 *Statistical Methodology*

The ultimate aim of GAME is to be able to mark assignments and to produce very similar results to those manually obtained by a human (commonly referred to as the 'Human Marker' in this research). In order to evaluate how well GAME performs, a correlation analysis was used to compare commenting, indentation, style, compilation and execution marks to those obtained by the Human Marker. A Pearson's linear correlation coefficient (r) [10] was used to measure the strength of association between the two sets of marks (one obtained by GAME and the other by the Human Marker), with r being 1

indicating a perfect correlation and 0 indicating no association. The significance level was set to 5%, so that if $p < 0.05$, the correlation would be found to be statistically significant, whereas if $p > 0.05$ the correlation would be found not to be statistically significant. However, if there was not a sufficient linear range to use a Pearson's Linear Correlation, then a sign test was used to determine whether the marks obtained by GAME match the Human Marker. This is based on a binomial distribution.

4. Results

The number of assessment items correctly marked by GAME is shown in Table 1 for Java Test 1 and Table 2 for Java Test 2 and for C++ Test 3. Table 3 shows the statistical results obtained for Java Test 1, Java Test 2 and C++ Test 3. Table 3 shows the test number, number of trials, total number of assignments, and correlation and p-values of results obtained from the GAME and Human Marker methods.

Table 1 Results of assignments marked by the GAME system that agreed 100% with the Human Marker for Java Test 1, trials 1, 2 and 3 combined.

Trial	Total Number Students	Structure Mark			Compile and Execute
		Commenting	Indentation	Style	
1	33	26 (78.8%)	30 (90.9%)	28 (84.8%)	27 (81.8%)
2	38	25 (65.8%)	34 (89.5%)	30 (78.9%)	32 (84.2%)
3	27	13 (48.1%)	27 (100%)	21 (77.8%)	18 (66.7%)
Total	98	64 (65.3%)	91 (92.8%)	79 (80.6%)	77 (78.6%)

Table 2 Results of assignments marked by the GAME system that agree 100 % with the Human Marker for Java Test 2 and C++ Test 3.

Trial	Total Number Students	Structure Mark			Compile	Execute
		Commenting	Indentation	Style	Correct	Correct
Java Test 2	21	19 (90.5%)	21 (100%)	17 (81.0%)	21 (100%)	12 (57.1%)
C++ Test 3	13	7 (100%)	7 (53.8%)	13 (100%)	13 (100%)	4 (30.7%)

Table 3 Statistical results obtained for Java Test 1, Java Test 2 and C++ Test 3

Test	Total number of assignments marked	Pearson Correlation Between Marks Obtained by GAME and the Human Marker Methods					
		Structure Mark			Compiles/ executes	Compiles	Executes
		Commenting	Indentation	Style	Correctly	Correctly	Correctly
Java Test 1 Trial 1	33	0.85 (p = 0.000)	0.85 (p = 0.000)	* p = 0.000, n = 33	0.90 (p = 0.000)	N/A	N/A
Java Test 1 Trial 2	38	0.43 (p = 0.007)	0.93 (p = 0.000)	* p = 0.003, n = 38	0.94 (p = 0.000)	N/A	N/A
Java Test 1 Trial 3	27	0.44 (p = 0.02)	1.00 (p = n/a)	* p = 0.013, n = 27	0.94 (p = 0.000)	N/A	N/A
Java Test 2	21	0.97 (p = 0.000)	1.00 (p = n/a)	0.64 (p = 0.002)	N/A	1 (p = 0.000)	0.79 (p = 0.000)
C++ Test 3	13	0.93 (p = 0.000)	0.49 (p = 0.088)	1.00 (p = 0.000)	N/A	1 (p = 0.000)	0.18 (p = 0.546)

* Indicates when a sign test was used due to a limited linear distribution

5. Analysis and Discussion

5.1 Java Test 1

Table 1 shows the first type of assessment item marked by GAME; a total of 98 assessment items were presented to the system. It can be seen from this table that GAME produced marks for Indentation, Style and Compile/Execution, which agreed 100% with the Human Marker in a high proportion of cases, although not as frequently as for Indentation. For Indentation these two sets of marks (i.e. GAME and the Human Marker respectively) were nearly identical in 90% or more of the cases: 91% in Trial 1, 90% in Trial 2 and 100% in Trial 3. However for Style and Compile/Execution the two sets of marks agreed around 80% of the cases. The statistical results shown in Table 3 also confirmed this observation. For Indentation and Compile/Execution a correlation coefficient (r) of greater than 0.85 was found. In all cases it was found that the correlations with the Human Marker were significant, as $p < 0.05$, indicating there was a

very close association between marks given by the GAME system and the Human Marker for Indentation and Compile/Execution of students' code.

For analysing Style, a Sign Test was used to determine whether the marks obtained by GAME matched the Human Marker. This was because the marking range was very narrow (in most cases a mark of 0 or 1.5 was given out of 5) and hence it was not possible to analyse the results using a Pearson's Linear Correlation. The statistical results shown in Table 3 indicated that two sets of marks matched significantly as a $p < 0.05$ in all cases for marking the Style of the students' code.

The marks for "Commenting" from GAME and the Human Marker agree in fewer instances than for "indentation", "style" and "compile/execute": 79% for Trial 1, 66% for Trial 2 and 48% for Trial 3 (Table 1). While these are sufficient to produce significant correlation coefficients for all trials (Table 3), these discrepancies prompted further investigation. Upon closer inspection of the raw data, it becomes clear that in cases where there are differences with regard to "commenting", GAME tends to award higher marks than the Human Marker. This can be explained largely by the fact that the GAME system, as it stands, is not yet able to distinguish what constitutes a "good" comment. In some cases the "comments" are in superseded code fragments that have been "commented out", so that they do not compile and as such they have no documentation value. In future, the GAME system can be improved by teaching GAME to distinguish what constitutes a "good" comment.

Closer examination of the raw data also helped to gain some understanding of the deficiencies of GAME in its present form. For example, with regards to compilation and execution, it was found that human error played a major part in the 19.4% of

discrepancies between GAME and the Human Marker. Out of the 19 assessments where conflict arose, 7 were related to Human Error. This underscores the obvious, but still noteworthy point that while human markers are the model for the automated marking system, they are by no means free from errors. The remaining 12 assessment items differed from GAME because the Human Marker quite often gave part marks for the student's effort. To overcome this problem in future, some additional strategies could be introduced into the GAME system to provide it with the ability to award part marks.

Overall, GAME was able to mark this simple assignment (containing one source file) quite well with respect to indentation, style and compilation/execution, as the marks were very closely related to the Human Marker. It was found to have a lower performance in marking commenting practice, by tending to give higher marks than the Human Marker. Further research is required to design more advanced marking strategies to improve these results.

5.2 Java Test 2

In Java Test 2, GAME was tested on a more difficult assignment that involves between 7 or 8 source files. Students were required to implement a command-line program with a specific number of menu choices. Only 21 out of the 23 assignments could be marked by GAME. In both of the remaining cases, the problem was due to the students altering the assessment criteria, by implementing different command-line menu structures to the one specified in the assessment guidelines.

From Table 2, in regards to indentation and compilation, it can be seen that GAME agreed 100% with the Human Marker, for all 21 of the assessment items marked. There was also nearly complete agreement between GAME and the Human Marker for

commenting and style, with respectively 19 (91%) and 17 (81%) of the assessments items correctly being marked. Closer inspection of assignments where GAME and the Human Marker did not agree with regards to style, revealed that where the Human Marker gave a mark of 2.5 out of 5, GAME tended to award a mark of 0. This indicated a lack of ability of GAME to award part marks, which was a commonly observed trait of the Human Marker. However, GAME did not perform well at marking the execution part of the assignment for JAVA Test 2, with only 12 (57%) of assignments agreeing with the Human Marker.

Statistical results in Table 3, validate the results discussed above. In all cases except style, GAME seemed to mark very closely to the Human Marker as a statistically significant correlation (r) of greater than 0.97 ($p = 0.000$) was obtained for commenting, indentation and compilation and an r of 0.79 ($p = 0.000$) for execution. Even with multiple source files, GAME seemed to perform very well at automated marking. The only problem encountered with GAME in JAVA Test 2 was style, attributed to the fact that the Human Marker awarded part marks.

For the 5 assessment items that disagreed with Human Marker for source code structure, one was for the commenting section and one was for the commenting and programming style sections. The Human Marker in both instances gave fewer marks for the commenting section due to the presence of some irrelevant comments in the student's code. The remaining three assessments that GAME disagreed with the Human Marker were for programming style, where the Human Marker assigned more marks than the GAME system.

For execution correctness, it was found that for the nine assessments that disagreed with the Human Marker, five were due to the students changing the nature of their assignment through alteration of either the menu structure or the order in which data was prompted for. For the other four assessments, the difference was attributed to the fact that the Human Marker gave part marks for the students' effort. This suggests that the discrepancies are not serious, in that more elaborate marking guides, some additional 'intelligent' component added to award part marks, would substantially increase the effectiveness of the GAME system.

Overall, the results suggest that GAME is sufficiently robust to handle multiple assignments with multiple, as well as single source files and different marking strategies.

5.3 C++ Test 3

The third major experiment devised to test GAME involved assignments from a first year C++ course. It was used to evaluate GAME's ability to mark other programming languages. A total of 14 students' assessments were fed to GAME, of which only 13 could be marked. The single case that could not be automatically marked, used code that disabled GAME's ability to pass the required input stream to the student's program.

The results in Table 2 revealed that GAME performed extremely well at marking the sections of commenting, style and compilation, as 100% of assignments marked by GAME agreed with the Human Marker. The statistical analyses (Table 3) also indicated the effectiveness of GAME as an automated marking method. For these sections, GAME was highly correlated with the Human Marker, as extremely statistically significant correlations of 1 ($p = 0.000$) were found. However, unlike Java Test 1 and 2 it seemed that GAME had difficulty marking indentation, as only 7 (53.8%) of assignments

agreed with the Human Marker. Statistical results (Table 3) validated this, as r was 0.49 ($p = 0.088$) indicating the correlation was poor and insignificant. One explanation for this result is that the multi-source file C++ assignments included some poorly formatted code from the instructor, which affected GAME's performance in this instance.

The ability for GAME to mark the execution of the assignment was also poor, this time only 4 (30.7%) of assignments agreed with the Human Marker. Statistical results (Table 3) showed the marks obtained by GAME in the execution section were not highly correlated with the Human Marker and were extremely insignificant with r being 0.18 ($p = 0.546$). From the raw data, this poor correlation seemed to be attributed to the fact that in all cases, the Human Marker awarded 5 out of 5 for execution, whereas GAME seemed to mostly always award zero. The low mark awarded by the GAME system could be tracked down to the fact that students compiled their projects in Visual C++, whereas, GAME used the command line compiler included in Visual C++. The command line compiler did not give global values an initial value (i.e. did not set a global integer variable to zero). This caused the large majority of assessment items to not execute properly, and therefore obtain a zero mark.

For C++ Test 3, the GAME system seemed to perform well at marking assignments written in a different programming language. Since GAME worked very well at marking indentation in Java Tests 1 and 2, the problems in C++ Test 3 were attributed to formatting problems with the instructor's code. The poor correlation between marks obtained by GAME and the Human Marker for execution, emphasizes that the GAME system must compile and execute students' source code in the same environment that the students work in to give consistent results.

6. Conclusions and Future Work

This paper presented an experimental analysis of GAME, an automatic system for marking programming assessment items, in university-level courses. The generic design of the system makes it a versatile tool for marking different types of assessment items. Overall, the results obtained from the experiments conducted suggest that the GAME system is very promising for use in a real-world teaching environment. It can handle submissions in different programming languages, which comprise of single or multiple source files. GAME also has the ability to accept and implement different marking strategies. Potentially, the advantages of employing an automated marking system like GAME are evident in terms of student-centered learning (ample and timely feedback) and objective, transparent mark allocation.

In analysing the experimental results, some interesting outcomes were found that might assist future research in this area. For example, in marking the correctness of a program's execution through an automated system like GAME, measures will need to be put in place for dealing with programs whose operations have diverged from those specified in the assignment sheet. This issue is being addressed in a web-based version of GAME (currently in development to investigate its role in facilitating student-centered learning) that will automatically reject assignments, which do not meet specifications. The students involved are thus given an early warning to modify their assignments to conform to the required specifications. Also, procedures for allocating part marks for a student's effort may need to be explored via the use of intelligent techniques. Finally, further marking strategies will be added to the system to deal with issues relating to the variety of output obtained from students' programs.

In its current form, GAME is a tool that is still being fine-tuned and used mainly for research and investigation only. As such, it is not yet available in the public domain. However, the authors' intention is that, once research is complete, GAME will be made publicly and freely available on the Internet.

7. Acknowledgments

The authors wish to thank the anonymous referees for a number of helpful comments, but retain full responsibility for any shortcomings, which may remain. This work was supported by Griffith University as part of its Teaching Grant scheme.

8. References

- [1] Malmi, L., Saikkonen, R. and Korhonen, A., 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), 55-59.
- [2] Jackson, D. and Usher, M., Grading student programs using ASSYST, *Proceedings of 28th ACM SIGCSE Tech. Symposium on Computer Science Education, San Jose, California, USA (1997)* 335-339.
- [3] Reek, K. A., The TRY system or how to avoid testing student programs, *Proceedings of SIGCSE 1989 (1989)*, 112-116.
- [4] English, J. and Siviter, P., 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), 168-171.
- [5] Saikkonen, R., Malmi, L. and Korhonen, A., 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), 133-136.
- [6] Jones, E. L., Grading Student Programs: A Software Testing Approach, *Journal of Computing in Small Colleges*, Vol. 16, No. 2, (2001), pp 185-192.
- [7] English, J., Automated Assessment of GUI Programs using JEWEL, *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*, (Leeds, UK, 2004), pp. 137-141.
- [8] Ghosh, M., Verma, B. and _____, __, An Automatic Assessment Marking and Plagiarism Detection, *First International Conference on Information Technology and Applications (ICITA 2002)*, (Bathurst, Australia 2002).
- [9] Blumenstein, M, Green, S., Nguyen, A. and Muthukkumarasamy, V., GAME: A Generic Automated Marking Environment for Programming Assessment, *International Conference on Information Technology: Coding and Computing (ITCC 2004)*, (Las Vegas, USA 2004), 212-216.
- [10] Blumenstein, M., Green, S., Nguyen, A. and Muthukkumarasamy, V., An Experimental Analysis of GAME: A Generic Automated Marking Environment, *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*, (Leeds, UK, 2004), pp. 67-71.
- [11] Zarr, J. H., *Biostatistical Analysis*. New Jersey: Prentice-Hall, (1999).