eGRADER
The Programming Solutions' Grader in Introductory Java Courses

Fatima AlShamsi and Ashraf Elnagar
Departement of Computer Science, College of Sciences, University of Sharjah, Sharjah, U.A.E.

Keywords: Java, Programming, Computer Science Education.

Abstract: This paper presents a graph-based grading system for Java introductory programming courses, eGrader. This system grades submission both dynamically and statically to ensure a complete and through grading job. While dynamic analysis is based on JUnit framework, the static analysis is based on the graph representation of the program and its quality which is measured by software metrics. The graph representation is based on the Control Dependence Graphs (CDG) and Method Call Dependencies (MCD). eGrader outperformed existing systems in two ways: the ability of grading submission with semantic-errors, effectively, and generating reports for students as a feedback on their performance and instructors on the overall performance of the class. eGrader is well received by instructors not only for saving time and effort but also for its high success rate represented by four performance measures which are sensitivity (97.37%), specificity (98.1%), precision (98.04%) and accuracy (97.07%).

1 INTRODUCTION

The idea of making the process of grading programming assignments automatic started with teaching programming. In 1960's, Hollingsworth (Hollingsworth, 1960) introduced one of the earliest systems which grade students programs written in Assembly language. Since then, the development and implementation of Automatic Programming Assignment Grading (APAG) systems has been a subject of great interest to many researchers. The need for decreasing the load of the work on the grader, timely feedback for the students and get rid of the emotional effects on the grading results are some of the reasons that motivated the need for APAG systems.

Although several automatic and semi-automatic programming grading systems were proposed in the literature, few of them can handle semantic errors in code. Besides, most of the existing systems are only concerned about the students' scores ignoring all other resulting data.

This paper presents a new system, eGrader, for grading Java students' solutions, both dynamically and statically, in introductory programming courses. Reports generated by eGrader make it a unique system not only to grade students' submissions and provide them with detailed feedback but also to assist instructors in constructing a database over all students and produce outcome analysis. In addition, eGrader is one of few systems to grade Java source code with the existence of semantic errors.

The remainder of the paper is organized as follows: Section 2 summarizes the existing APAG systems. Section 3 discusses the methodology adopted in eGrader. Components of eGrader framework are described in Section 4. In Section 5, we discuss the experimental results. We conclude the work and present possible future directions in Section 6.

2 RELATED WORK

Different approaches have been adopted to develop APAG systems. Approaches can be categorized to three basic categories; dynamic or test based, semantic-similarity based, and graph based.

The dynamic-based is the most well known approach that has been used by many existing systems. Douce *et al.* reviewed automatic programming assessments which are dynamic-based in (Douce et al., 2005). Using this approach, the mark assigned to a programming assignment depends on the output results from testing it against a predefined set of data. However, this approach is

AlShamsi F. and Elnagar A..
eGRADER - The Programming Solutions' Grader in Introductory Java Courses.
DOI: 10.5220/0003338600360045
In *Proceedings of the 3rd International Conference on Computer Supported Education* (CSEDU-2011), pages 36-45
ISBN: 978-989-8425-50-8

not applicable if a programming assignment does not compile and run to produce an output. In this case, no matter how the assignment is good it will receive a zero mark. Moreover, using dynamic-based approach does not ensure that the assignment producing correct output is following the required criteria. Examples of dynamic-based systems are Kassandra (Von Matt, 1994) and RoboProf (Daly & Waldron, 2004).

The semantic similarity-based (SS-APAG) approach overcomes the drawbacks of the dynamic-based approach. Using this approach the grading of a student's program is achieved by calculating semantic similarities between the student's program and each correct model program after they are standardized. This approach evaluates how close a student's source code to a correct solution? However, this approach can become expensive in terms of time and memory requirements if the program size and problem complexity increase. ELP (Troung et al., 2002) and SSBG (Wang et al., 2007) are two examples of this approach.

The graph based approach is a promising one which overcomes the drawbacks of other approaches. This approach represents source code as a graph with edges representing dependencies between different components of the program. Graph representation provides abstract information that is not only supports comparing source codes with lower cost (than semantic similarity approach) but also enables assessing source code quality through analyzing software metrics. Comparing graph representations for two programs is done on the structure level of the program. This approach has been applied in two different ways: graph transformation such as in (Truong, 2004) and graph similarity such as in (Naude, 2010).

3 METHODOLGY

eGrader can efficiently and accurately grade a Java source code using both dynamic and static analysis. The dynamic analysis process is carried out using the JUnit framework (Massol & Husted, 2003) which is proved to be effective, complete and precise. It provides features that do not only ease the dynamic analysis process but also makes it flexible to generate dynamic tests for different types of problems in several ways.

The static analysis process consists of two parts: the structural-similarity which is based on the graph representation of the program and the quality which is measured by software metrics. The graph representation is based on the Control Dependence Graphs (CDG) and Method Call Dependencies (MCD) which are constructed from the abstract syntax tree of the source code. From the graph representation, structure and software metrics are specified along with control structures' positions and represented as a code which we call it *Identification Pattern*. The result of static analysis is the output of the matching process between students' identification pattern and models' identification patterns.

3.1 Identification Pattern

The identification pattern is a representation of the structure and software engineering metrics of a program. The structure is presented in the identification pattern based on the program tracing (without executing it) starting from the *main* method. The structure and software engineering are two major components of any identification pattern.

3.1.1 The Structure Component

Table 1: Basic Categories and Controls of the structure component of the identification pattern.

Basic Category	Code	Control	Code
Conditions	1	if_statement	1
		elseif_statement	2
		else_statement	3
		switch_statement	4
		case_statement	5
		General_statement	*
Loops	2	for_loop	1
		while_loop	2
		dowhile_loop	3
		General_loop	*
Method calls	3	Recursive method call	1
		Non recursive method call	2
		General_method_call	*
Exceptions	4	try_block	1
		catch_block	2
		finally_block	3
		General_block	4

The structure component consists of several sub components represented with a mask of digits. Each sub-component represents a control structure or a method call in the program structure. Each sub-component is composed of three types of codes: basic category, control and position.

Table 1 shows the code representation for basic categories and controls of the structure components.

For example, a for loop control is of the `Loops` basic category and `for_loop` control which is represented with the code `21`. The code `1*` is a representation for the `Conditions` basic category and `General_statement` control, which means any of the `Conditions` control is acceptable. This type of coding is used in the model solution's programs only.

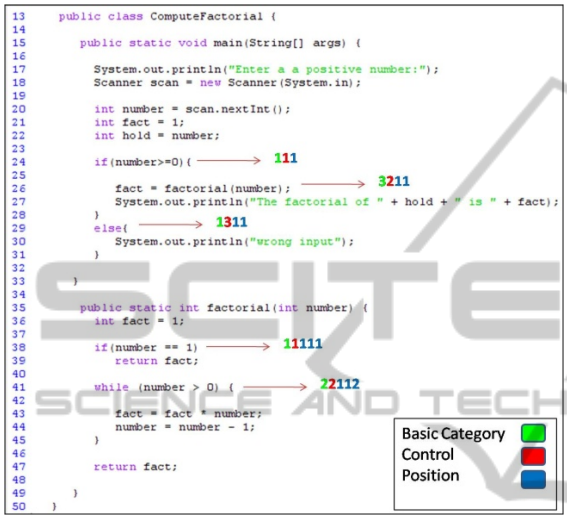


Figure 1: ComputeFactorial class.

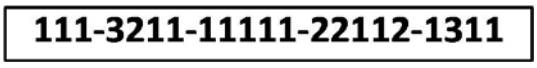


Figure 2: Structure component of `ComputeFactorial` class.

The position code consists of one or more digits representing the position of a control structure or a method call in the whole program structure. It also represents the position relative to other control structures and method calls in the program structure.

Figure 1 depicts an example of the structure component for `ComputeFactorial's` identification pattern. Class `ComputeFactorial` in Figure calls the method factorial to compute the factorial value after checking its validity `(number >=0)`.

To trace `ComputeFactorial`, we start with the control structure `if (number >= 0)`. Since this control structure is a condition control of type if_statement, the basic category is set to `1` and the control is set to `1` too. The position of this control structure is `1` as it's the first control structure to trace. The second control structure to trace is the method call `fact = factorial(number)` which is a call to a non recursive method. The basic category for the method call is `3` and a non recursive method has the control value `2`. Since `fact = factorial(number)` is control dependent on the first control structure to trace, which is `if (number >= 0)`, the number of digits in the position code will increase by one and will be `11`. The if_statement at line `38` inside the method factorial has the code `11111`, where the first `1` is for the basic category (conditions), the second `1` is for the control (if_statement) and `111` is for the position. The control structure `while (number > 0)` at line `41` is traced after the control structure at line `38`, so `while (number > 0)` has a position value greater than the position value of `if (number >= 0)` by one which is `112`. The else_statement at line `29` is the last control structure to trace and it is control dependent on `if (number >= 0)`. The code for else is `1311`, where `1` is for the basic category, `3` is for the control and `11` is for the position.

The whole ordered structure component of `ComputeFactorial`'s identification pattern is shown in Figure 2.

3.1.2 Software Engineering Metrics (SEM) Component

Software Engineering Metrics (SEM) consist of 3 sub-components. Each sub component represents one of the three SEM respectively, number of variables, number of classes and number of library method calls. Each sub component consists of two or three parts depending on whether the SEM component if for a student's program or a model program.

For student's program each sub-component consists of two parts: Basic category and Number. The basic category codes are 5 for Variables, 6 for Classes, and 7 for Library method calls. The Number represents the number of each SEM component in the student's program.

For the model program, each sub-component consists of three parts: The Basic category, MinNumber and MaxNumber. The basic category coding follows the same strategy as in student's program SEM component. Parameters MinNumber and MaxNumber consist of two digits each representing the minimum and the maximum number of SEM sub-component allowed, respectively.

507-601-704

Figure 3: A student's SEM component of Figure 1.

50407-60101-70410

Figure 4: A model's SEM component of Figure 1.

Figures 3 and 4 show examples of SEM component for identification patterns. An example of a SEM component for `ComputeFactorial` (Figure 1) as a student's program is shown in Figure 3. The basic category of type Variables has a number set to `07` which means the student used 7 variables in his/her program. The code `601` means there is one class in the file. The number of library method calls in the student's program is `04` which is represented in the code `704`, where 7 indicates the basic category (type library method calls). An example of a SEM component for `ComputeFactorial` of Figure 1 as a model program is shown in Figure 4. The basic category of type Variables has a MinNumber equals to `04` and MaxNumber equals to `07` meaning that students are allowed to use a minimum of 4 variables and a maximum of 7 variables. Students should not use more than one class which is represented by the code `60101`. The code `70410` indicates that students are allowed to use a minimum of 4 library method calls and no more than `10`, where 7 represents the basic category of type library method calls.

3.1.3 Structure and SEM Analysis

The main idea behind the identification pattern is to analyze both the structure and the SEM of students' programs. Therefore, an efficient strategy to compare identification patterns is required. Certain criteria need to be met to develop an efficient strategy to compare identification pattern. The criteria are as follows:

1. Identification pattern matching is based on the distance between them. The distance measure used is the number of missing control structures and SEM for the model program in addition to the number of extra control structures and SEM in the student's identification pattern.

$$D = | N_{Missing} + N_{Extra} | \quad (1)$$

Where D is the distance, $N_{Missing}$ is the number of missing control structures, and N_{Extra} is the number of extra control structures.

2. If there exists a model identification pattern that matches exactly a student's identification pattern, the distance between both is set to zero.
3. If no exact match found, the best match is the model's identification pattern which has the minimum distance D with the student's identification pattern.
4. If two models' identification patterns have the same distance from the student's identification pattern, the best match is the one that maximizes the scored mark.
5. The maximum distance equals the number of control structures and SEM in the model's identification pattern in addition to the number of control structures and SEM in the student's identification pattern. No match exists if this criterion is valid for all models' identification pattern given a student's identification pattern.

```
12  public class ComputeFactorial {
13
14      public static void main(String[] args) {
15
16          System.out.println("Enter a positive number: ");
17
18          Scanner scan = new Scanner(System.in);
19
20          int number = scan.nextInt();
21
22          if (number >= 0) {
23
24              System.out.println("The factorial of "
25              + number + " is " + factorial(number));
26
27          }
28          else{
29
30              System.out.println("wrong input");
31
32          }
33      }
34      public static int factorial(int number) {
35
36          if (number <= 1) {
37
38              return 1;
39          }
40          else {
41
42              return number * factorial(number - 1);
43          }
44      }
45  }
```

111-3111-11111-131111-3111111-1311-50407-60101-70407

Figure 5: Recursive solution.

To illustrate our comparison process, an example for calculating factorial is presented. This example consists of two models' solution and one student's solution. The first model solution calculates factorial using a recursive method (Figure 5). The second one is nonrecursive solution (Figure 6). An example of a student's solution is shown in Figure 7.

```
12      public class ComputeFactorial {
13
14       public static void main(String[] args) {
15
16         System.out.println("Enter a a positive number:");
17         Scanner scan = new Scanner(System.in);
18
19         int number = scan.nextInt();
20         int fact = 1;
21         int hold = number;
22
23         if(number>=0){
24
25            fact = factorial(number);
26            System.out.println("The factorial of "
27            + hold + " is " + fact);
28         }
29         else{
30            System.out.println("wrong input");
31         }
32      }
33       public static int factorial(int number) {
34         int fact = 1;
35
36         while (number > 0) {
37
38            fact = fact * number;
39            number = number - 1;
40         }
41
42         return fact;
43
44      }
45   }
```

111-3211-2*111-1311-50710-60101-70407

Figure 6: Non recursive solution.

```
12 public class ComputeFactorial {
13
14    public static void main(String[] args) {
15
16       int theNum=1, theFact=1;
17
18       Scanner input = new Scanner(System.in);
19
20       System.out.println("This program "+
21       "computes the factorial of a number.");
22
23       System.out.print("Enter a number: ");
24       theNum = input.nextInt();
25
26       theFact = factorial(theNum);
27
28       System.out.println(theNum +
29        "! = " + theFact + ".");
30    }
31
32    public static int factorial(int n) {
33
34       if (n <= 1) {
35
36          return 1;
37
38       } else {
39
40          return n * factorial(n - 1);
41
42       }
43    }
44
45 }
```

311-1111-13111-311111-505-601-704

Figure 7: A student's solution.

The student's identification pattern is compared with the first model identification pattern in Figure 8. The basic category and control of each control structure in the student's identification pattern is compared with the basic category and control of each control structure in the model's identification pattern until a match is found. The distance D in this example is equal to 2, as two control structures are missing;

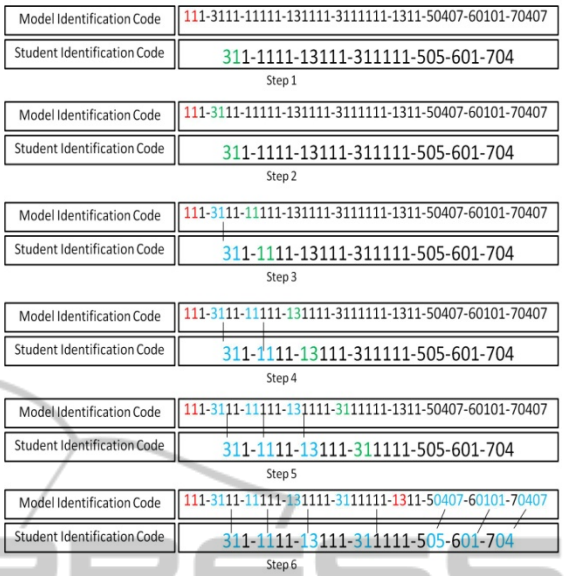


Figure 8: Comparison process between the student's solution in Figure 7 and the model solution in Figure 5.

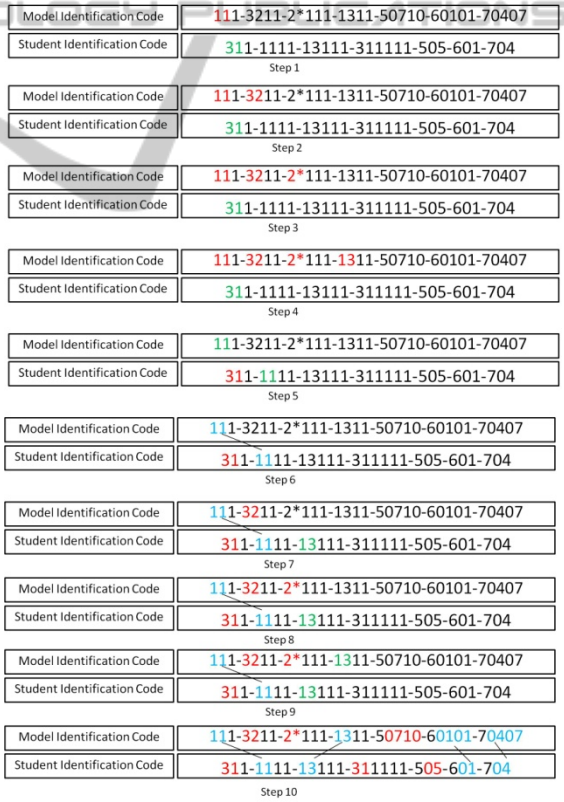


Figure 9: Comparison process between student's solution in Figure 7 and model solution in Figure 8.

if_statement and elseif_statement.

In Figure 9, the student's identification pattern is compared with the second model's identification

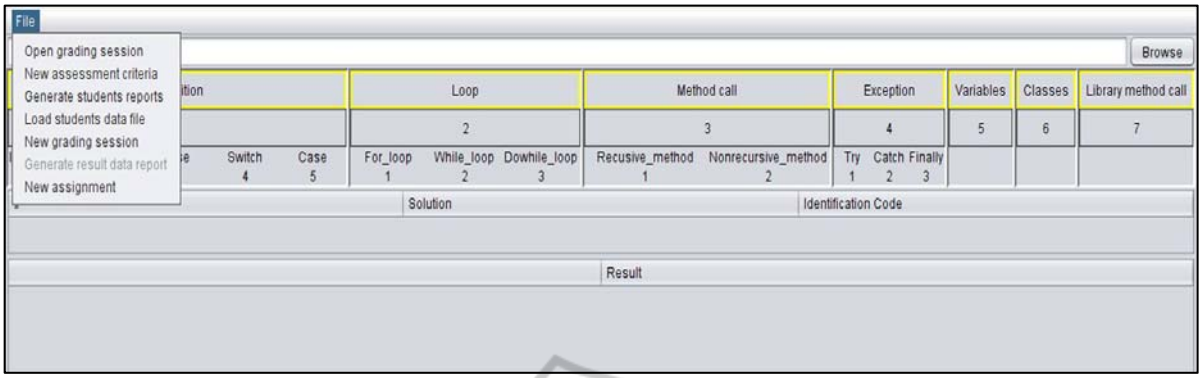


Figure 10: eGrader basic screen.

pattern. Steps 1 to 4 show that no matching is found for the control structure `311` of the student's identification pattern. The comparison process proceeds to the next control structure in the student's identification pattern which is `1111` at Step 5. The result of the comparison at Step 10 indicates 2 extra control structures, 2 missing control structures and 1 missing SEM where `505` doesn't match `50710`. Therefore, the distance D is equal to 5.

As a result, the first model's identification pattern better matches student's identification pattern than the second one. The mark is to be assigned based on the first model program.

4 eGRADER FRAMEWORK

The framework of eGrader consists of three components: Grading Session Generator, Source code Grader, and Reports Generator. eGrader basic screen in shown in Figure 10.

4.1 Grading Session Generator

eGrader supports both generating and saving grading sessions. Generating a grading session is easy, flexible and quick. A grading session is generated through three steps: creating model list, creating assessment criteria, and creating new grading session.

4.1.1 Creating Model List

Figure 11 shows the flow chart for Creating Model List Component. Model list is created simply by adding model solutions, where Identification Patterns (IP) and Software Engineering Metrics (SW) are generated automatically. Once an identification pattern is generated, a dialog box appears showing the identification pattern and providing a possibility to modify it. The modification options are: to choose another form of Java control structures or a general form.

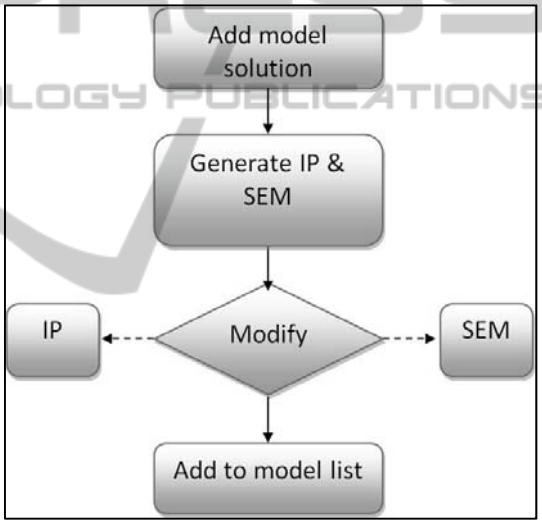


Figure 11: Flow chart of Creating Model List Component.

SW metrics are optional. Such metrics include number of variables, number of library methods and number of classes used. Adding each of the SW metrics along with their values to the IP is optional.

The model identification code is added then to a list that can be saved and modified at another time.

4.1.2 Creating Assessment Criteria

Assessment criteria are categorized into five categories:

A. Condition Statements.
B. Loop Statements.
C. Recursive & Nonrecursive method calls.
D. Exceptions.
E. Variables, classes and library method calls.

Each category provides input fields for measuring category weight and penalty (except for category E) for extra controls. A category is added to the grading process if it has a weight greater than zero. If penalty value of a category is greater than zero, a student who used extra controls (more than required in the program) of that category will be penalized. Weights and penalty values are normalized. Options in each category's check list covers all the controls in an introductory Java course. Assessment criteria can be saved for later use.

4.1.3 Creating New Grading Session

A grading session is created through *New grading session dialog*. In this dialog three files need to be added which are the solutions set file, the assessment criteria file and the JUnit test file with an option for specifying the weight (which has to be in the range of [0-1]) for dynamic analysis phase. Other files can be included such as data files to run or test students' submissions.

4.2 Source code Grader

As most of the existing systems do, the submitted source code need to be a zipped file named with the student's identification number. This naming and submitting strategy is chosen in order not to burden the instructor with both searching for required files in different folders and keeping track of which submission belongs to which student. The grading process steps are as follows:

1. Loading grading session. List of solutions will be loaded, directories and identification pattern in a table form in the main eGrader's frame.
2. Loading the submitted zipped files by specifying their folder.
3. Submissions will be graded and their output will be inserted into a table.

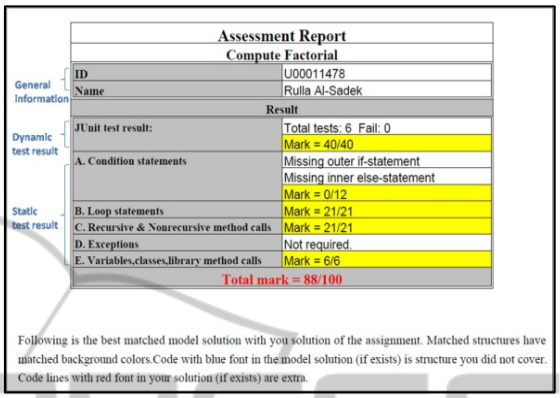
At this stage, the grading process is completed. The list of students' names along with their details is kept in excel file that is to be loaded to eGrader.

4.3 Reports Generator

eGrader not only grades Java code effectively but also provides the instructor with detailed information about the grading process. It helps to analyze students' understanding of basic programming concepts. There are two types of reports are produced by eGrader: students assessment reports and class reports.

4.3.1 Students Assessment Reports

After the grading process is completed and the student data file is loaded, students' reports are generated.



	Assessment Report	
	Compute Factorial	
General Information	**ID**	U00011478
	Name	Rulla Al-Sadek
	Result	
Dynamic test result	**JUnit test result:**	Total tests: 6 Fail: 0
		Mark = 40/40
Static test result	**A. Condition statements**	Missing outer if-statement
		Missing inner else-statement
		Mark = 0/12
	B. Loop statements	Mark = 21/21
	C. Recursive & Nonrecursive method calls	Mark = 21/21
	D. Exceptions	Not required.
	E. Variables,classes,library method calls	Mark = 6/6
	Total mark = 88/100	

Following is the best matched model solution with you solution of the assignment. Matched structures have matched background colors.Code with blue font in the model solution (if exists) is structure you did not cover. Code lines with red font in your solution (if exists) are extra.

Figure 12: Result (First and second sections) of a student's report for Computer Factorial assignment.

Student assessment report is a report produced for each student that consists of four different sections:

1. Identification: contains student information such as name, identification number, the result of grading his/her submission. Figure 12 shows an example for Compute Factorial assignment.
2. Marking: shows the details of the marking scheme after conducting both the dynamic and static tests. The dynamic test result includes the total number of tests and the number of tests that failed. The static part shows the 5 general categories and the mark for each one, if required. In the case of encountering errors, a message will be inserted to indicate the source of this error. Marks are deducted based on the original marking scheme set by the instructor/grader. Example is shown in Figure 12.
3. Model solution: points to the model solution that best matches student's submission. Example is shown in Figure 13.
4. Original code: shows students' solution. A matching between the structure of the model solution and the structure of the student's submission is displayed using color matching between corresponding control structures. Example is shown in Figure 14.

Your solution

```
import java.util.Scanner;
public class ComputeFactorial {
    public static void main(String[] args)  {
        int number;
        int fact = 1;
        System.out.println("Please enter number to calculate it's factorial");
        Scanner input = new Scanner(System.in);
        number = input.nextInt();
        fact = factorial(number);
        System.out.printf("%d factorial is %d\n", number, fact);
    }
    public static int factorial(int number)  {
        int fact = 1;
        for (int i = 1; i <= number; i++)  {
            fact = fact * i;
        }
        return fact;
    }
}
```

Figure 13: Model solution (third section) of a student's assessment report.

Model solution

```
package ;

import factorial.*;
import java.util.Scanner;
public class ComputeFactorial {
    public static void main(String[] args)  {
        System.out.println("Enter a positive number:");
        Scanner scan = new Scanner(System.in);
        int number = scan.nextInt();
        int fact = 1;
        int hold = number;
        if (number >= 0)
            {
                fact = factorial(number);
                System.out.println("The factorial of " + hold + " is " + fact);
            }
else
            {
                System.out.println("wrong input");
            }
    }
    public static int factorial(int number)  {
        int fact = 1;
        while (number > 0)  {
            fact = fact * number;
            number = number - 1;
        }
        return fact;
    }
}
```

Figure 14: Student solution (fourth section) of a student's assessment report.

A report for a student's submission that contains syntax errors consists of one part only, which indicates that the submission has syntax errors and to be checked by a grader. The total mark for this submission is zero. An example is shown in Figure 15.

Assessment Report	
Compute Factorial	
ID	U00011123
Name	Nora Mohammed Ahmed
Result	
Grading submission failed....	Solution contains syntax error(s).To be graded by the instructor.
Total mark=0/100	

Figure 15: A student´s assessment report for a submission containing syntax errors.

4.3.2 Class Reports

A class report is a summary report on the class performance for a specific assignment. This report consists of three parts (three excel sheets) which are: statistics, dynamic test details and static test details.

	A	B
1	Detailed outcome for assignment Compute Factorial	
2	Number of submissions	14
3	Number of model solutions	9
4	Most popular model solution	C:\Users\Fatima\Desktop\dataset\factorial\s1\ComputeFactorial.java
5	Least popular model solution	C:\Users\Fatima\Desktop\dataset\factorial\s6\ComputeFactorial.java
6	Number of unit tests	6
7	Number of submission failed all unit tests (dynamic tests)	2
8	Number of failed submissions (syntax errors)	1

Figure 16: Statistics part of Compute Factorial assignment report.

Useful information such as the assignment's difficulty level, the number of students who managed to submit a solution, and the most and least common solutions, can be derived from the statistics part.

As presented in Figure 16, the statistics part contains the following data:

- Number of students' submissions for a given assignment based on the number of graded submissions.
- Number of model solutions used to grade the submissions.
- Most popular model solution.
- Least popular model solution.
- Number of unit tests used to test submission which is taken from running JUnit test class against a model solution.
- Number of submissions failed all unit tests. This number indicates the submissions that failed all the tests in the JUnit test class.

	A	B
1	Tests failed	Number of student failed the test
2	testFactorial1	1
3	testFactorial	4
4	testFactorial3	4
5	testFactorial4	4
6	testFactorial5	4
7	Common failures	
8	5! : expected:<120> but was:<24>	
9	12! : expected:<479001600> but was:<39916800>	
10	5! : expected:<120> but was:<15>	
11	3! : expected:<6> but was:<3>	
12	12! : expected:<479001600> but was:<46080>	
13	3! : expected:<6> but was:<2>	
14	10! : expected:<3628800> but was:<3840>	
15	10! : expected:<3628800> but was:<362880>	
16	0! : expected:<1> but was:<0>	
17		
18	Average mark	29.04761905
19	Maximum mark	40
20	Minimum mark	13.33333333

Figure 17: Dynamic test details part of Compute Factorial assignment report.

	A	B
1	**Assignment Requirements**	
2	A. Condition statements	
3	Average mark	9
4	Maximum mark	12
5	Minimum mark	0
6		
7	B. Loop statements	
8	Average mark	20.7
9	Maximum mark	21
10	Minimum mark	18.9
11		
12	C. Recursive & Nonrecursive method calls	
13	Average mark	20.85
14	Maximum mark	21
15	Minimum mark	18.9
16		
17	D. Exceptions	
18	Not Required	
19		
20		
21		
22	E. Variables,classes,library method calls	
23	Average mark	6
24	Maximum mark	6
25	Minimum mark	6
26		
27	Average total	88.02821
28	Maximum total	100
29	Minimum total	67.33333

Figure 18: Static test details part of Compute Factorial assignment report.

- Number of failed submissions because of syntax errors.

The dynamic test details part provides a general overview of the performance of the class. This part is shown in Figure 17. It displays the following data:

- Tests failed along with the number of students failed each test.
- List of runtime errors. Such information is useful for the instructor to identify common problems and as a result provide necessary clarification of some concepts in class.
- Other useful statistics such as average, maximum and minimum marks.

Static test details part provides information on the performance of the class in the general five categories. This part as shown in Figure 18 consists of the following data:

- Assignment Requirements which contains five categories, where each has three measures: average mark, highest mark and lowest mark, if the category is required. Otherwise, the category will be reported as not required. Group A. Condition statements; for example, is represented by, the average mark which is the average of all submissions marks for this group, the highest mark which is the highest submission's mark for this group and the lowest mark which is the lowest submission's mark for this group. The same applies for all the other categories.
- Other useful statistics such as average, maximum and minimum marks.

5 EXPERIMENTAL RESULTS

eGrader has been evaluated by a representative data set of students' solution in Java introductory programming courses at the University of Sharjah. This data set consists of students' submissions for two semesters with a total of 191 submissions with an average of 24 students in each class. The assignment set covers 9 different problems.

Four types of programming assignments were used, which are:

- Assignment_1: tests the ability to use variables, input statements, Java expressions and mathematical computations and output statements.
- Assignment_2: tests the ability to use condition control structures such as if/else-if/else and switch and case statement. It also tests students' abilities to use loop structures such as for, while and do-while statements.
- Assignment_3: tests the ability to use recursive and non recursive methods.
- Assignment_4: tests the ability to use arrays.

We are using four performance measures to evaluate eGrader performance. Namely, sensitivity, specificity, precision and accuracy.

Sensitivity measures how many of the correct submissions are in fact rewarded. Whereas the specificity is a measure of how many of the wrong submissions are penalized. Precision is a measure how many of the rewarded submissions are correct. Finally, accuracy is a measure of the number of correctly classified submissions.

Evaluation shows a high success rate represented by the performance measures which are sensitivity (97.37%), specificity (98.1%), precision (98.04%) and accuracy (97.07%) as shown in Figure 19.

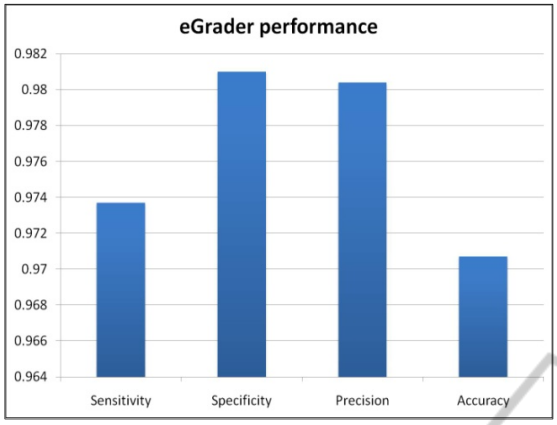


Figure 19: eGrader performance.

6 CONCLUSIONS AND FUTURE WORK

eGrader is a graph based grading system for Java introductory programming courses. It grades submissions both statically and dynamically to ensure a complete and through testing. Dynamic analysis in our approach is based on the JUnit framework which has been proved to be effective, complete and precise. This makes it a suitable tool for the problem of dynamic analysis for students' programs. The static analysis process consists of two parts: the structure-similarity which is based on the graph representation of the program and the quality which is measured by software metrics. The graph representation is based on the Control Dependence Graphs (CDG) and Method Call Dependencies (MCD) which are constructed from the abstract syntax tree of the source code. From the graph representation, structure and software metrics are specified along with control structures' positions and represented as a code which we call it Identification Pattern.

eGrader outperformed other systems in two ways. It can efficiently and accurately grade submissions with semantic error. It also generates a detailed feedback for each student and a report for the overall performance for each assignment. This makes eGrader not only an efficient grading system but also a data mining tool to analyze students' performance.

eGrader was appraised by instructors and teaching assistants for its overall performance (97.6%) and the great reduction in time needed for grading submissions when using it. Their comments provided useful feedback for improvement.

eGrader can be extended to incorporate other features such as:

- Support GUI-based programs.
- Grade assignments in other programming languages.
- Offer the eGrader online.

REFERENCES

Daly, C. & Waldron, J., 2004. Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. New York, 2004. ACM.

Douce, C., Livingstone, D. & Orwell, J., 2005. Automatic test-based assessment of programming: a review. *Journal on Educational Resources in Computing*, 5(3), p.4.

Hollingsworth, J., 1960. Automatic graders for programming classes. *Communications of the ACM*, 3(10), pp.528-29.

Massol, V. & Husted, T., 2003. *JUnit in Action*. Greenwich, CT, USA: Manning Publications Co.

Naude, K.A.a.G.J.H.a.V.D., 2010. Marking student programs using graph similarity. *Computers & Education*, 54(2), pp.545-61.

Troung, N., Bancroft, P. & Roe, P., 2002. ELP--A Web Environment for Learning to Program. In *The 19lh Annual Conference of the Australian Society for Computers in Learning in Tertiary Education*. Auckland, 2002.

Truong, N.a.R.P.a.B.P., 2004. Static analysis of students' Java programs. In *Proceedings of the sixth conference on Australasian computing education-Volume 30*. Dunedin, New Zealand , 2004. Australian Computer Society, Inc.

Von Matt, U., 1994. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22, pp.22-26.

Wang, T., Su, X., Wang, Y. & Ma, P., 2007. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2), pp.99-107.