JavaMarker: A Marking System for Java Programs

Marzieh Ahmadzadeh
School of Computer Engineering & IT
Shiraz University of Technology
Shiraz, Iran

Sahar Namvar
School of Computer Engineering & IT
Shiraz University of Technology
Shiraz, Iran

Mansoore Soltani
School of Computer Engineering & IT
Shiraz University of Technology
Shiraz, Iran

ABSTRACT

In this paper a marking system for Java programming is presented which has been developed as a plug-in for a widely used editor, *Eclipse.* This system runs student submitted programs against previously defined test cases. Depending on the percentage of correct running code, a proper mark is awarded. Since this program was implemented in order to be used in a principles of programming course, we require students to practice coding with a correct style. Therefore, this system checks the style of the code and produces messages when a better style is expected. In some cases penalty marks are considered for improper code style.
For this system to play an educational role, we allow students to submit more than once. With this we aim to help them learn from their mistakes. The number of submissions differs from one exercise to another and is defined dynamically by our system administration.
We call this system JavaMarker.

General Terms

Design, Experimentation, Languages.

Keywords

Eclipse, Plug-in, Java, Marking System.

1. INTRODUCTION

Learning to program is a learning-by-doing task which requires an excessive effort of course administration to run the course efficiently. One of the important, and also difficult, tasks is to mark huge numbers of assignments and to provide effective feedback in a timely manner. However, this is not always possible because of the high number of students and the broad range of content which requires frequent exercises. Therefore, course administrators are rarely able to mark students' programs and provide feedback when and where it is necessary. One solution to speed the process up is to increase the number of teaching assistants. This creates the problem of inconsistency in marking. Although one can plan to assign one exercise to one teaching assistant and another exercise to another teaching assistant, still an inconsistency in overall marking can be observed. Adding to that, it is not possible to provide feedback exactly when it is required.

Further, there are some very simple concepts in terms of the quality of the code that we want our students to get used to. But they do not take it seriously because no compiler error is issued if they do not care about that style. For example, we emphasize that indentation is important since it makes the program more readable. We have observed cases where all the lines of the code start from the same column even in late semester. Therefore, we looked for an opportunity to enforce the use of a correct style. This also requires instant feedback. Even if we have to reduce marks in the early stages of programming, students will learn to pay attention to code quality the next time around.

All of these issues motivated us to develop a system that marks students programs automatically. From an educational point of view we would have the ability to provide instant feedback and from the administrative side we would assure that marking is consistent throughout the semester.

Of course several programs have been introduced in literature to mark students' programming exercises [6, 7, 8]. In these systems, developers have implemented an editor from scratch. Perhaps the focus on qualities such as correctness of the program has prevented an implementation of a comprehensive editor.

Since one of our educational objectives is to prepare students to enter industry, we aim to prepare them as much as possible and do not want them to get surprised when they start working just because they have not seen professional editors previously. Therefore, implementing our system in the form of a plug-in for a professional editor seemed a good idea.

Fortunately, there exists *Eclipse* [3], which is widely used for programming in Java and provides a large amount of facilitating tools that helps programmers to focus on the task of programming. Further, adding developed plug-ins is allowed.

This persuaded us to implement an *Eclipse* plug-in for our intended system.

What will be seen in this paper is as follows. In section two related works are explained. The strategy that we used in our marking is described in section three. Our way of incorporating the strategy in our developed system is explained in section four. Future work is discussed in section five.

2. RELATED WORKS

To be able to implement our marking system, we needed to look for strategies that were used in literature. We came across two sets of related works.

First were the developments of marking systems that have spanned nearly two decades [2, 4, 5, 7, 9]. These systems share the same properties. They were developed to evaluate C or Java programming language and they all evaluated programs by both comparing the outputs with predefined sets of test data and the quality of code in terms of style. Of course, there existed some other systems that were implemented in order to evaluate programs written in other languages such as Scheme [10].

Our aim was to enforce using real world programming editors. This not only helps students to do their projects in later courses such as Software Engineering easier, but also prepares them to work in industry. Therefore, works that were introduced in literature were of little help. In all the reviewed works either a simple editor was implemented and integrated with the evaluation system [2] or, where the use of existing systems were required, the marking systems were not integrated [3,4]. Therefore, these systems were deemed to be ad hoc.

Second were the strategies that were needed to mark students' programs for which two issues needed to be considered; marking the execution of the program and the quality of the code.

The fact that the nature of assignments can be completely different from each other forced the developer of such systems to implement the strategy of comparing the generated results by a program against expected outputs character by character. This approach, which has been used by most of these introduced systems, is similar to our method of assessing the correctness of a program.

There were extensive reports on the metrics that were used for assessing program quality in the literature [1, 2, 11]. We used these metrics in our system and have explained them in section 3.1.

3. MARKING STRATEGY

In our programming course in which Java language is taught, two criteria are considered to award marks to students. The first criterion is concerned with submitting a working code. This depends on the proportion of the code that outputs correct results. If all segments of the code run correctly, a full mark is given otherwise, a percentage of the mark is awarded. The second criterion is the quality of the code in terms of programming style.

In order to develop an automated marking system which marks students' programs similar to human marking strategies, we considered both criteria in our plug-in. Our incorporated strategy is discussed in section 3.1 and 3.2.

3.1 Metrics for Program Quality

Several issues were considered for program quality based on previous research [1, 2, 11] in addition to some other factors that we thought we must habituate novice students to use. After collecting all the cases, it was decided that not all of these issues should contribute to students' marks but they are still of such importance that we wanted to point them out for educational purposes. Therefore, we divided the style cases into two groups. The first group contained instances that resulted in mark reduction if they were excluded from the program. For the second group no mark penalty was considered but a warning message was issued to notify students of not regarding that special case.

To mark the code style, we designed a table in our database in which the optimum number of each style issue was entered. For example, one of our considerations was the total number of defined methods for a specific assignment. Therefore, we have a field in our table that shows the interval that is accepted (i.e. minimum and maximum number of defined methods). If the number of methods was in that interval, a full mark was awarded; otherwise, a percentage of the mark was given. The computation of this percentage comes from Benford [2].

Parameters such as *depth of inheritance*, *total number of children of a class*, *the number of defined methods*, *number of variables defined in a method*, *number of loops*, *size of methods*, *number of global variables*, *number of defined and unused methods*, [2], *incorrect blank or break line* [2, 11], number of private methods, number of final classes and number of static methods play a role in having a proper code style. These parameters are included in the first group that we award/reduce marks for. As stated previously, the computation of the mark is based on the optimum numbers that are specified in our table.

The type of comments used (i.e. use of // or /*), providing initial comment, incorrect blank line (excluding before and after methods) and incorrect indentation are among the second group of parameters [11]. Therefore, a warning message is issued and no mark is reduced.

3.2 Metrics to Measure a Correct Code

To measure the correctness of a program, we designed several test cases for each assignment to evaluate students' programs against. We considered different numbers of test cases depending on the type of assignment because some programs needed more testing than others. This included cases where the program should have run normally plus all possible cases where we expected the program to report an error.

The idea of preventing students from unlimited numbers of submissions was first introduced by Reek [9] in order to enforce in students to think about what went wrong in their program. In this way, random modifications by students to get the right result could be avoided.

The strategy is to divide the total mark depending on the number of outputs. For example, if the program generates 5 outputs and each of the five has the same importance in weight, we consider 20% of the total for each output. Since these percentages are defined dynamically for each assignment, different weighting is also possible.

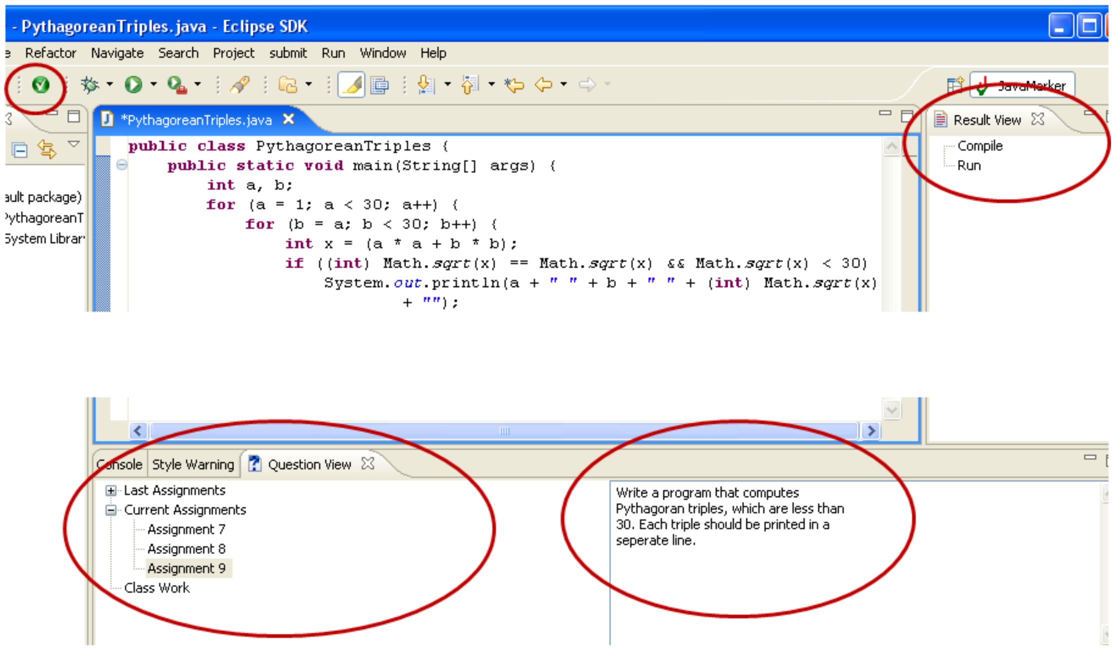
Another issue that we considered was the state in which we awarded the mark. If a program ran up to one point and then stopped due to the occurrence of an exception, we gave marks up to that point and issued the exception to let students know that their program did not end normally.

If the program did not output all the required outputs, just a percentage of the total mark related to correct output was awarded. This included the situation when an infinite loop was generated and the program needed to be timed out.

If a program was submitted with a compiler error, we reported the proper message and no mark was awarded.

4. ECLIPSE PLUG-IN: JavaMarker

For JavaMarker to function effectively for our current and future requirements, a careful design was needed. We divided the explanation of this system into three parts. The first part, which is hidden from the user's view, explains the architecture of the system and is discussed in part 4.1. The second part shows the perspectives that are created by the system and viewed by the users. This part is shown in section 4.2. The final part, section 4.3, describes what happens when students submit their programs.



```
public class PythagoreanTriples {
    public static void main(String[] args) {
        int a, b;
        for (a = 1; a < 30; a++) {
            for (b = a; b < 30; b++) {
                int x = (a * a + b * b);
                if ((int) Math.sqrt(x) == Math.sqrt(x) && Math.sqrt(x) < 30)
                    System.out.println(a + " " + b + " " + (int) Math.sqrt(x)
                            + "");
```

Write a program that computes Pythagoran triples, which are less than 30. Each triple should be printed in a seperate line.

Figure 1: A sample view of JavaMarker

4.1 Architecture

The marking system consists of two parts; a database and facilitating code for database connection plus, the actual plug-in code. The database and related code were designed to exist on the server while the plug-in runs from the client side. When a program is submitted, it is first saved temporarily on the client side. It is then executed (i.e. compiled and run) by our plug-in. And finally the code, resulting mark and other related information is sent to a table in our database.

The tables were designed in such a way that the whole system can run flexibly and efficiently. To have such a flexible system several issues were involved.

Among these issues was the problem of transforming the configurations of laboratory computers from time to time. Therefore, the path that a student’s program was temporarily saved could not be statically defined. For this the path information was read from a table in the database. In case we were not allowed to use a specific path, we were able to allocate a new path by specifying it in the corresponding table.

Another issue was the number of allowed submissions, which differed from one exercise to another. For early assignments we considered more submissions to let students learn the important issues that allocate marks to themselves. In later assignments, the number of submissions was reduced. The allowed number of submissions is specified in a table. Each row in the table corresponds to one assignment.

Sometimes in our course we encourage pair programming, therefore the same mark is awarded to two students of the same group. Since every two students log in with the same account, the information regarding a group’s name and account is also kept in a table to enable us to award marks to both of them.

As stated before, students are given marks based on the percentage of correct output. Some outputs are more important than others. Therefore, the allocated percentage varies from one output to another. For this, each of the test cases was divided into parts and a percentage of the mark was assigned to them. This is the information that is read from the table when a program is tested against our test cases.

A further important issue was a time for program time out if an indefinite loop is encountered. This can also be set dynamically by assigning a field of database table.

Since one assignment can be completely different from another assignment, it should be taken into account that the process of marking cannot be static. For example, we are not able to allocate 10% of the style mark for the *depth of inheritance* in all of our assignments because most of the early programs do not contain any inheritance relations. Instead, in early programs, issues such as *correct indentation* would have more importance. For this, we used a kind of weighting strategy and formed a table in which attributes are our defined metrics for program quality in terms of style issues. Each row of this table belongs to one of our assignments and the value of attributes shows the percentage of the total style mark that is allocated to the attribute. Thus, if a value of an attribute such as *depth of inheritance* was set to zero, no mark would have been awarded for this attribute. This of course is dependent on what is expected from the students. For example, a program might be written efficiently if a hierarchy of inheritance is considered. However, the weight of this attribute is set to zero if inheritance has not been taught yet.

All of the cases that were explained in section 3.1 have different weights in different programs. These weights are specified separately for each assignment in the related designated tables.

4.2 Perspectives

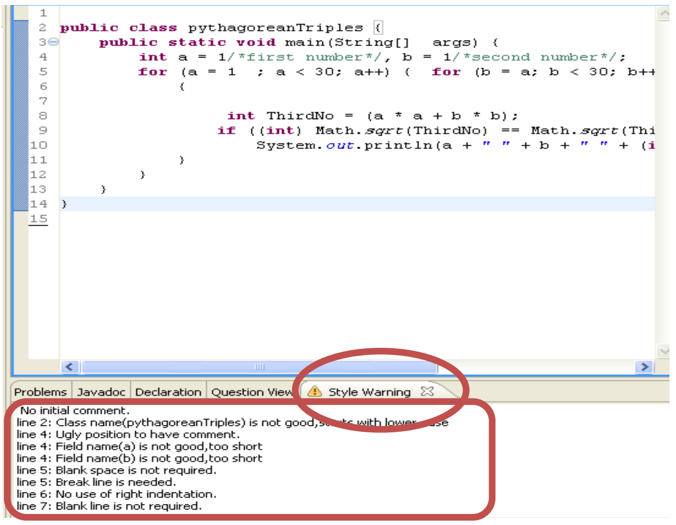
As can be seen in Figure 1, JavaMarker creates several views.

A new menu item has appeared on the upper left of the screen which is the *submit* button. When students think that their program is ready to be submitted, they only need to click this button.

The right hand view is where students can see their mark. If the program has no compiler error, the awarded mark can be seen under the ‘Run’ part. If some compiler errors exist, the proper message is shown under the ‘Compiler’ part and no mark is awarded. If the program runs incompletely (i.e. it works for some test cases but not for others or the execution stops abnormally due to the existence of an exception), a percentage of the mark is granted and a description is given regarding the reasons for the mark reduction. Again the related mark is shown under the ‘Run’ part.

Underneath the code view, ‘Question View’ can be seen in which assignment numbers are shown in a hierarchical representation. Clicking on one of the assignments, program specification is shown on the right hand side of the question view.

As explained earlier, in the right hand view of the editor, the awarded mark can be seen. This mark corresponds to both the execution of the program and code style. Since the message regarding the execution of the program is short (i.e. 50% test case 1, 25% test case 2), the message is given in the same view. However messages relating to program style are more descriptive and need more space. Therefore, a separate view is considered for exhibiting those comments. This view is shown in Figure 2.



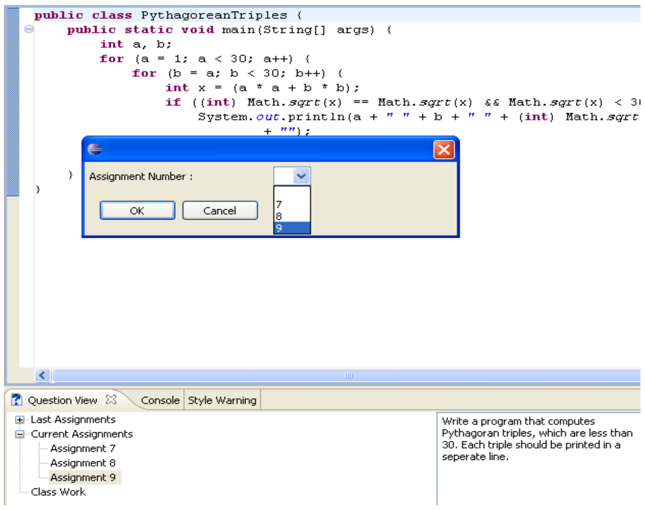
```
 1
 2 public class pythagoreanTriples {
 3     public static void main(String[]  args) {
 4         int a = 1/*first number*/, b = 1/*second number*/;
 5         for (a = 1  ; a < 30; a++) {  for (b = a; b < 30; b++
 6             {
 7
 8                  int ThirdNo = (a * a + b * b);
 9                 if ((int) Math.sqrt(ThirdNo) == Math.sqrt(Thi
10                     System.out.println(a + " " + b + " " + (i
11             }
12         }
13     }
14 }
15
```

```
 No initial comment.
line 2: Class name(pythagoreanTriples) is not good,starts with lower case
line 4: Ugly position to have comment.
line 4: Field name(a) is not good,too short
line 4: Field name(b) is not good,too short
line 5: Blank space is not required.
line 5: Break line is needed.
line 6: No use of right indentation.
line 7: Blank line is not required.
```

Figure 2: A sample view to comment on the style of the code

4.3 Execution

It is time to submit the code when our student is satisfied with his/her program and thinks that the program is error free. Clicking on the submit button, a list of exercises is shown and he/she selects the exercise that he/she intends to submit. Of course the exercises that passed the deadline are excluded from the list. The reason that we provide such a list is that some students are quicker than others so we allow them to deliver their code well ahead of the deadline. This way they can start or even submit the next exercise sooner. As can be seen from Figure 3 only exercises 7 to 9 were allowed for submission at the time of providing this snapshot.



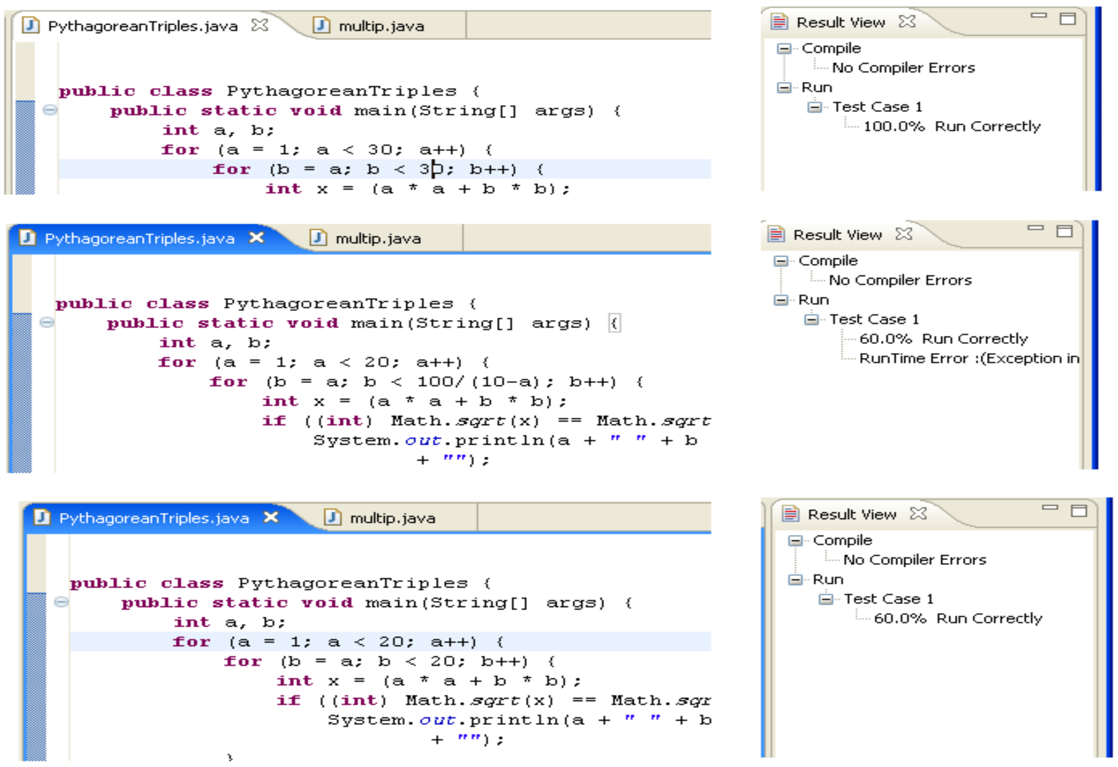
```
public class PythagoreanTriples {
    public static void main(String[] args) {
        int a, b;
        for (a = 1; a < 30; a++) {
            for (b = a; b < 30; b++) {
                int x = (a * a + b * b);
                if ((int) Math.sqrt(x) == Math.sqrt(x) && Math.sqrt(x) < 3
                    System.out.println(a + " " + b + " " + (int) Math.sqrt
                            + "");
    }
}
```

Write a program that computes Pythagoran triples, which are less than 30. Each triple should be printed in a seperate line.

Figure 3: A sample view to submit an assignment

Before accepting students’ codes, our program checks the total number of allowed submissions and our student’s number of submissions from the related table. If he/she is allowed to submit, the program is accepted. Then the information of the test cases, including input, corresponding output, the percentage of allocated mark to each output and to each test case, is sent to the client side from our database which is in the server. After examining the code and granting the mark, all of the information, including source code and corresponding mark, is sent to the database.

Figure 4 shows the execution of three slightly different programs. Since these programs do not require an input and only generate a specific output, only one test case is sufficient. The first program was correct. All of the output matched with the expected ones therefore, 100% was awarded. The second case was run up to a point but stopped due to the presence of an exception. Therefore, the related mark (60%) was awarded and a message regarding the presence of an exception issued. Sixty percent of the mark was granted to the third program since only 60% of outputs were generated.



```
public class PythagoreanTriples {
    public static void main(String[] args) {
        int a, b;
        for (a = 1; a < 30; a++) {
            for (b = a; b < 30; b++) {
                int x = (a * a + b * b);
```

```
Compile
    No Compiler Errors
Run
    Test Case 1
        100.0%  Run Correctly
```

```
public class PythagoreanTriples {
    public static void main(String[] args) {
        int a, b;
        for (a = 1; a < 20; a++) {
            for (b = a; b < 100/(10-a); b++) {
                int x = (a * a + b * b);
                if ((int) Math.sqrt(x) == Math.sqrt
                    System.out.println(a + " " + b
                        + "");
```

```
Compile
    No Compiler Errors
Run
    Test Case 1
        60.0%  Run Correctly
        RunTime Error :(Exception in
```

```
public class PythagoreanTriples {
    public static void main(String[] args) {
        int a, b;
        for (a = 1; a < 20; a++) {
            for (b = a; b < 20; b++) {
                int x = (a * a + b * b);
                if ((int) Math.sqrt(x) == Math.sqr
                    System.out.println(a + " " + b
                        + "");
            }
```

```
Compile
    No Compiler Errors
Run
    Test Case 1
        60.0%  Run Correctly
```

Figure 4: A view of the execution of different programs

5. CONCLUSION & FUTURE WORK

Since this program is still in the early stages of its life, there are some issues that must be resolved in the future.

For instance, it is normal that we set some constraints for carrying out some exercises to ensure that students practice what we ask them to. For example, we might ask them to check the divisibility of a number to a special number n. We might also emphasize that they are not allowed to use remainder operator because we want them to practice working with loops. Another example is that we might ask them to solve a problem without using *stack*, but they can use an array and treat it as a stack, which is not what we desire. These are some implications that our program is not able to manage at the moment.

Also at the time of this writing, students are required to have all their *classes* in a single file.

In addition, to test students' programs based on our arbitrary input, we would like to have a mechanism to evaluate program correctness for any program that does not input a data. There are some situations in which an input is not provided from an input device. Instead, variables get initialized by a programmer. For instance, a program may generate values for an array and then order the array. We would like to see whether the ordering is done correctly. The only way to do this is to ask students to initial the variables with what we tell them to enable us to match the generated output to the expected one. In the future we have to design a mechanism to measure the correctness of outputs independent of a variable's initial value.

Another limitation of our current program is the lack of ability to grade the process with which the program is written. We would like to see that our novice programmer has a relatively correct view of the problem in mind before he/she starts to code. This can be easily recognized when you observe students write their code. But we found it fairly difficult to incorporate in the marking system because there are a huge number of considerations involved.

Further, our current program is not able to mark programs that include graphical features.

Finally we would like to add a plagiarism detection tool to our plug-in to create a relatively full educational package.

6. REFERENCES

[1] Al-Ja'afer, J., and Sabri, K. 2005. Automark++ a Case Tool to Automatically Mark Student Java Programs. *The International Arab Journal of Information Technology*, Vol. 2, No. 1, January 2005

[2] Benford, S., Burke, E., Foxley, E. and Higgins, C. 1995. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual Southeast regional conference* (ACM-SE 33). ACM, New York, NY, USA, 176-182.

[3] Clayberg, E. and Rubel, D. 2008, Eclipse Plug-ins. 3rd Edition by Addison-Wesley Professional.

[4] Daly, C. AND Waldron, J. 2004. Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education.* 210-213.

[5] Daly, C. 1999 RoboProf and an Introductory Computer Programming Course. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*. 155-158.

[6] Jackson, D. AND Usher, M. 1997. Grading Student Programming Using ASSYST. In *Technical Symposium on Computer Science Education, Proceedings of the 28th SIGCSE* (San Jose, CA), 335-339.

[7] Joy, M., Griffiths, N., and Boyatt, R., 2005.The BOSS Online Submission and Assessment System., ACM Journal on Education Resources in Computing, vol.5, No.3, September 2005, Article 2.

[8] Higgins, C., Hegazy, T., Symeonidis, P., AND Tsintsifas, A. 2003. The CourseMaster CBA system:Improvements over Ceilidh*. J. Edu. Inf.Technol. 8*, 3, 287-304.

[9] Reek, K. A. 1989. The TRY system – or – how to avoid testing student programs. *SIGCSE Bull. 21*, 1, 112-116.

[10] Saikkonen, R., Malmi, L., AND Korhonen, A. 2001. Fully automatic assessment of programming exercises. In *Proceedings of the ITiCSE 2001Conference*, ACM Press, New York, 133-136.

[11] Sun Microsystem. April 20, 1999. Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/codeconventions-150003.pdf. Last Accessed on 6/1/2011