Automatic Grading of Student’s Assignments

Rudolf Pecinovský
University of Economics Prague
Department of Information Technologies
Churchill sq. 4
130 00 Prague 3

rudolf@pecinovsky.cz

ABSTRACT
One of the bothersome tasks in programming education is evaluation of student assignment solutions and homeworks, because this activity is time consuming and mostly not interesting. The paper shows how the evaluation of handed in solutions can be easily automated and introduces a “micro library” used for this purpose. Then it shows how we can modify the evaluation process to minimize problems caused by handing in solutions prepared by somebody else.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*.

General Terms
Management, Verification

Keywords
Automated grading, Interfaces, Java Software Engineering Education, Design-Patterns-First, teaching theory.

1. INTRODUCTION
The preparation of student assignments and especially subsequent evaluation of their solutions belongs to the less pleasant parts of the teacher’s activities. If all the students had the same assignments, then there would be a danger that the advanced students would transmit their solutions to their colleagues, who could make several simple replacements to make the discovery of the transmission difficult. On the other hand different assignments for each student complicate the evaluation of their solutions.

The classical evaluation technique is relatively laborious and often boring: the teacher has to run each program, give it some data and check that the program reacts as required. Each such evaluation can take time in the order of minutes. Rarely do we clip this evaluation time to a few seconds.

Therefore automatic and semiautomatic grading of student programs has been of interest to computer science educators (see e.g. [2], [3], [5], [6], [7]). However, there seems to be no established consensus on the best way to automatically grade student code.

By preparation of an automatic grading system the ability to use the interface-driven assignments brings great advantage. [4] shows how this advantage can be used by preparing and evaluating assignments in advanced courses. But [9] and [8] showed that if we use the *Design Patterns First* teaching methodology we can use the automated grading even from the second lesson of the introductory course.

College of informatics and statistics at Economic University Prague invites almost 700 students pro year. They all have to pass the *Introduction to Programming* and *Introduction to Software Engineering* courses. Therefore it is very important for us to have some efficient system for automatic evaluation of homework as well as the larger assignments.

2. GRADING PROCESS
Let’s have a look at our system. We start with out favourite saying:

The program that is almost running
is like the plane that is almost flying.

In other words: we evaluate only such parts of handed in solutions that are really running. It is not important how large a part of the project was developed. The only work that the student can defend is the part of program that actually runs.

As we said above, the automated evaluation of the handed in solutions is significantly simpler when one of the requested features is an implementation of a given interface. In such a case the compiler checks the keeping of one part of assignment (class and method signatures) for us and we can concentrate on the proper evaluation of the solution.

We use for the automated evaluation of handed in solutions the microlibrary containing one public interface and one public class:

- The interface **`IGrader<T>`** characterises classes, where we define the test and possibly grade of the handed in solutions. The interface declares only one method **`int grade(T)`**. Its parameter is the graded/tested object that is an instance of a class implementing the interface **`T`** or class extending the class **`T`**. This method tests the obtained instance, possibly writes somewhere the message reporting results of their execution, but primarily it returns the grading of this instance.

```
public interface IGrader<T> {
    int grade( T gradedInstnace );
}
```

- The searching class **`Searcher<T>`** whose instances have to find all the classes whose instances should be graded (or only tested), create their instances and pass them for grading (or testing). The constructor of this class has two parameters:
  - **`Class<T> parent`** – a class-object of the interface that should be implemented by graded/tested classes or a class-object of the class that should be extended by graded/tested classes,
  - **`IGrader<T> grader`** – an instance of the grading class; this instance should be able to grade/test the obtained objects.

```
public class Searcher<T> {
    public <T> Searcher( Class<T>   parent,
                         IGrader<T> grader )
    { /* Constructor body */ }

    public void apply()   { /* Method body */ }

    //... Remaining code
}
```

We run the search for graded/tested classes and their grading by calling the method

`apply()`

It searches the folder (package) with the grading class and all its subfolders for classes implementing or extending the **`parent`**, creates an instance of each found class and passes this instance for grading/testing to the **`grader`**.

After obtaining the grade, it asks the graded instance for its author and puts the obtained grade into author's record in the database. To enable this second part of work (writing the result in the database), we define the class or interface **`parent`** as a child of interface **`IAuthor`** declaring two methods:¨

- **`String getAuthor()`** – it returns the author's name and surname for quick identification by human,
- **`String getXname()`** – it returns the unique student's ID identifying the particular student in all school databases.

```
public interface IAuthor {
    String getAuthor();
    String getXname();
}
```

So we are sure, that each graded instance is able to identify its author.

As you can deduce, the only activity that takes some time is the creating of the grading class. Students save their solutions into a given package/folder (in larger projects each one into subpackage of his/her own) and then it remains for us only to run the prepared grading program that checks and evaluates everything. The proper evaluation is then a few seconds (or for complicated evaluations a few minutes) task.

3. VARIANT ASSIGNMENTS

It's a known fact that assignments common to the whole class invite the less experienced students to copy the solution from the more experienced ones. We can simply protect against these favourite vices by different assignments for each student or for small groups of students. The common property of all assignments is the implemented interface or parent class.

It could seem at first look that different assignments will make the evaluation more complicated. By appropriately selecting assignments this complication is negligible. We can generate a great number of different assignments as a combination of several basic simple assignments. This great number needn't necessarily significantly complicate the evaluation and grading. We can prepare a small grading method for each basic assignment and by grading the whole solution recognize (e.g. from student's ID) which particular basic assignments were combined

Other possible ways to solve the difference in assignments is to give to every assignment an identifier. In the implemented interface or parent class we declare an abstract method for returning the identifier of the solved assignments. Our grading class can be supplemented by an auxiliary class, which offers a method returning the list of grading steps corresponding to the assignment, which identifier we passed to this method as the argument. An example of such a solution is presented e.g. in [9].

4. FREE PROJECTS

The above explained solution is easily applicable even on assignments with very free definition. We should only set a common framework defining some interfaces allowing future testing and grading.

An example of such an assignment can be a conversational game that our students should develop at the end of the first course (it is inspired by a similar game presented in [1]). In this game the player is going through a virtual world, asks the computer some questions and according to its responses he decides how to continue. The only rules that the game should fulfil are:

- The player should go through some rooms (or their equivalents: planets, education degrees, career levels etc.).
- There are some objects in the room; the player should collect some of them to use for carrying out some tasks in the future.
- The number of objects that the player can carry at one time is limited.

As you can see, the assignment is relatively general and it is very difficult to find a way to test all the solutions uniformly without spending a lot of time by communicating with applications through a keyboard.

However when we supplement this general assignment with a framework (see fig. 1) to which the student's solutions should be connected and cooperate with them, we get an interface enabling us to prepare the grading class and subsequently use our **`Searcher`**.

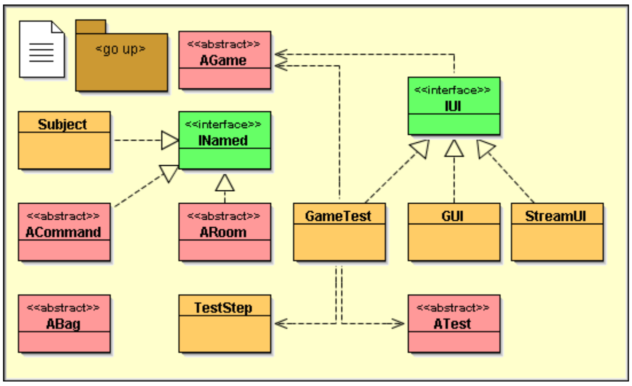

Figure 1: Frame for project Game

So we have a framework defining borders and rules. For testing and grading we need also a set of tests. However we cannot design these tests for all students because we don't know what game each particular student will design and program. So we divide the whole solution into two stages:

- In the first stage the student should hand in a scenario of their game. This scenario should be designed as an instance of a class extending the abstract class `ATest` implementing `Iterable<TestStep>` where `TestStep` defines a given set of important characteristics determining the game state after each particular step. This scenario will serve as test for the program prepared in the second stage.
- In the second stage the students hand in a package with a project meeting the previously handed in scenario/test.

The grading of such handed in solutions is again very simple.

5. TAKING OF FOREIGN SOLUTIONS

Now we would like to touch on taking of the solution from colleagues or ordering a complete solution for money. We have to admit that preventing students from the taking of foreign solutions is complicated and in homeworks almost impossible. Therefore we think that it is more profitable to accept this possibility and modify the system of handing in solutions in order to force students to study them.

We solve this problem in such a way that we announce, that we don't care about the source of the handed in solution, but we want the students to know this solution as well as if they designed it alone. The submission of the solution is therefore connected with a duty to modify the solution in a given way or to correct some artificially created error and force the modified or repaired solution to again pass the tests. This process we call the defence of the handed in solution.

We explain to students that almost every programmer takes at some time a foreign solution, however only the gambler incorporates in his/her program a module, whose functionality he/she has to guarantee without understanding it.

Our experience has shown that many students believe that it is sufficient to let the author explain the functionality of the program the night before its presentation. From this explanation they get the feeling that they understand the program, but they don't realize that understanding the explanation is a significantly different level of understanding than is the level needed for successfully modifying the program.

Every year therefore it happens that some students underestimate this difference. They bring their solution, but when they come to modify it a little, they wonderingly discover that the program they understood yesterday evening and supposed it to be clear, is now strange and full of un-understandable constructions that they are not able to correct or modify.

Students soon became aware that just handing in the foreign solution is not the right way to pass the exam and that learning how to modify such a solution is easier when they cooperate in its development.

6. CONCLUSION

The paper has shown that the appropriate use of the interface puts in our hands a very powerful tool for designing assignments and especially for following grading of the handed in solutions. It has demonstrated how to prepare assignments forcing students to master the skills that they will need in their work and simultaneously allowing the teacher to automate the evaluation of the handed in solutions.

The article also introduces the first version of the library used for automation of the grading of handed in programs. Its use can significantly shorten the time needed for this evaluation.

Finally it explained how we force students to understand their handed in solutions and be able to modify them even if they hand in solutions developed by colleagues or ordered for money.

7. REFERENCES

[1] BARNES David J., KÖLLING Michael: *Objects First With Java: A Practical Introduction Using BlueJ (2nd edition).* Prentice Hall, 2004. ISBN 0-131-24933-9.

[2] EDWARDS Stephen H.: Teaching software testing: automatic grading meets test-first coding. *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, 2003.

[3] HARRIS Archer J., ADAMS Elizabeth S., HARRIS Nancy L.: *Making program grading easier: but not totally automatic.* J. Comput. Small Coll., 20(1):248–261, 2004.

[4] HELMICK Michael T.: Interface-based Programming Assignments and Automatic Grading Java Programs, *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*, University of Dundee 2007.

[5] JACKSON David, USHER Michelle: Grading student programs using ASSYST. In SIGCSE '97: *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 335–339, New York, NY, USA, 1997. ACM Press.

[6] MALMI Lauri, KORHONEN Ari, SAIKKONEN Riku: Experiences in automatic assessment on mass courses and issues for designing virtual courses. In ITiCSE '02: *Proceedings of the 7th annual conference on Innovation and*

technology in computer science education, pages 55–59, New York, NY, USA, 2002. ACM Press.

[7] MALMI Lauri, KORHONEN Ari, SAIKKONEN Riku: Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, ITiCSE'01, pages 133–136, Canterbury, United Kingdom, 2001. ACM.

[8] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila: Order of Explanation Should be Interface – Abstract Classes – Overriding, *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*, University of Dundee 2007.

[9] PECINOVSKÝ Rudolf, PAVLÍČKOVÁ Jarmila, PAVLÍČEK Luboš: Let's Modify the Objects First Approach into Design Patterns First, *Proceedings of the Eleventh Annual Conference on Innovation and Technology in Computer Science Education,* University of Bologna 2006.