Evaluation of Student Assignments

Rudolf Pecinovský
University of Economics Prague
Department of Information Technologies
Churchill sq. 4
130 00 Prague 3
rudolf@pecinovsky.cz
+420 603 330 090

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 we can automate the evaluation of handed in solutions by using *Design Patterns First* methodology. It first sums up the basic characteristic of the *Design Patterns First* methodology and shows, why its use facilitates the designing of the assignments and subsequent evaluation of handed in solutions. It informs about a "micro library" used by author for this purpose. In the next part it shows using three examples taken from a basic course of Java how it is possible to make the evaluation significantly more effective.

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

General Terms
Design, Languages

Keywords
Objects First, Design Patterns, 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 the all students had the same assignments, then there is a danger that the advanced students will transmit their solutions to their colleagues, who can 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 standard evaluation technique is relatively laborious and often boring: the teacher has to run every program, give it some data and check that the program reacts as required. Each of such evaluations can take time in the order of minutes. Rarely do we clip this evaluation time to a few seconds.

Some teachers therefore try to prepare assignments, whose solutions can be easily evaluated. However this attempt often ends with assignments asking for result values written somewhere they can be "checked by eye". It still remains necessary to run each program and look somewhere at the results. The evaluation time therefore is linearly dependent on the number of evaluated solutions.

It has certainly occurred to many readers that a programmer could automate repeated and frequent tasks. However, for such automated evaluation we need to collect all solutions in one place and then discover the "key classes" from the auxiliary ones and ask the "key classes" for their answers in the right way.

For common types of assignments as met in textbooks and courses such automation is therefore reasonably solvable only for assignments requesting one source code and communication through standard input and output. Once we want a more complicated solution our automated evaluation can meet some problems.

As I signalled in the abstract, when we apply the *Design Patterns First* methodology, we have got the resources that allow us to automate evaluation of even relatively complicated solutions from almost the very beginning. Let's have a look at these resources and the mentioned methodology in a little greater detail.

2. BASIC PRINCIPLES OF METHODOLOGY *DESIGN PATTERNS FIRST*
The reason for developing the methodology *Design Patterns First* was the inability of other methodologies to keep some basic pedagogical principles, especially the *Early Bird* pedagogical pattern ([1]) that says: "*Organize the course so that the most important topics are taught first.*"

2.1 Interface
One of the basic principles of modern programming is *Programming to an Interface, not an Implementation*. By applying this rule to programming in Java the construct *interface* is very often used. However other methodologies explain it somewhere near the end of the first or even second course. So the students don't have sufficient opportunity to really learn it and no time to digest it and to learn how to incorporate it into their designs.

The methodology *Design Patterns First* incorporates the explanation of the interface into the first few lessons. The early incorporation of this subject offers students not only sufficient area to closely familiarize with it but simultaneously it offers teachers broad possibilities of how to prepare assignments whose evaluation can be automated. These possibilities are not achievable using other methodologies.

2.2 Design patterns

The next noticeable feature of modern programming is an intensive use of design patterns. Design patterns came to public awareness only in the second half of the nineties after [4] was published. Their acceptance grew very quickly and now their knowledge is considered one of the programmers' key skills.

The very early introduction of design patterns and its intensive use both in various examples and in assignments is the second main principle of the described methodology. This principle is considered so important that it gives its name to the whole methodology.

2.3 Test driven development

The third principle that this methodology aims to instil to students from the very beginning is the importance of tests and the profitability of their definitions even before the development of the tested program.

In early lessons the teacher formulates assessments as the test classes with tests for the required solution. In later lessons the preparation of tests is the first step in solving problems. The students can not start the next step before the teacher agrees with the scenario of the planed program defined by the test.

Using this approach students very soon meet the advantages of the programming style whose only goal is to fulfil the tests prepared in advance. This increases the probability that students embrace this progressive method and will use it in their future work.

3. UTILIZING OF THESE PRINCIPLES BY CREATING ASSIGNMENTS AND NEXT EVALUATION OF SOLUTIONS

Let's have a look how the early explanation of the above mentioned principles can be projected into homework and other assignments. I start with my 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 really runs.

In the early lessons I prepared assignments containing a test class with a set of tests the solution has to fulfil. The students' task is to define the class implementing the given interface and fulfilling all tests.

The fact that instances of classes created by students complete the test proves the correctness of the handed in solution. However it doesn't facilitate the evaluation very much. There remains the duty to run the test for every solution and find out the degree of fulfilment of the requirements.

We could automate the evaluation of handed in programs by defining some rules for class names and preparing a program to search for classes with such class names, create their instances and test the behaviour of these instances.

The weak point of such automation is the remarkable inability of students to keep even the simple conventions. We need to set "conventions" checkable by compiler or test program. Therefore we would incorporate in the checking program a part that checks for the keeping of these conventions. Fortunately this task is relatively simply solvable.

The automated evaluation of the handed in solutions is significantly simpler at the moment, when one of the requested features is an implementation of the given interface, because in such a case the compiler checks the keeping of this convention for us and we can concentrate on the proper evaluation of the solution.

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

- The interface `ITest<T>` characterises classes, where we define the test of the handed in solutions. The interface declares only one method `test(T)`. Its parameter is the 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 and writes somewhere the message describing results of this testing.
- The searching class `Tester` whose instances have to find all the classes whose instances should be tested, create their instances and pass them for testing. The constructor of this class has two parameters:
  - a class-object of the interface that should be implemented by tested classes or a class-object of the class that should be extended by tested classes,
  - an instance of the testing class that is able to test the obtained objects.

  We run the search for classes for testing by calling the method `verify(Class<T>,ITest<T>)` that searches the project folders for tested classes, creates an instance of each found class and passes this instance for testing to the object obtained as the second parameter.

So the only activity that takes some time is the creating of the test class. Students save their solutions into a given package/folder (possibly each one into subpackage of his/her own) and then it remains for us only to run the prepared test program that checks and evaluates everything. The proper evaluation is then a few seconds (or for bigger number of students minutes) task.

4. VARIANT ASSIGNMENTS AND AUTOHOR IDENTIFICATION

It's a known fact that a common assignment invites the less experienced students to copy the solution from the more experienced ones. We can simply defend 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 the first look that different assignments will make the evaluation more complicated. By the appropriately selecting assignments this complication is negligible. One of the 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 test class can be supplemented by an auxiliary class, which offers a method of obtaining an identifier of

the solved assignments and it returns a set of tests verifying the handed in solution.

In a similar way we can solve the solution's author identification. The implemented interface or parent class can declare an abstract method for returning the name (or some other identifier) of the author of this solution. By this identifier we are able to identify not only the author of the given solution, but also that the student solved the right version of the assignment.

5. TAKING OF FOREIGN SOLUTION

Now I digress from the subject for a while and touch on taking of the solution from colleagues or order 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 I think, that it is more profitable to accept this possibility and modify the way of handing in solutions in order to force students to study the handed in solution.

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

I 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.

My 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 came 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.

6. EXAMPLES

Let's return to assignments and evaluation of their solutions.

6.1 Five

The first task is creating a simple graphical object consisting of several other graphical objects. Such a graphical object can be e.g. the classical dice. The students get three classes:

- The interface that should be implemented by their solutions and that declares simple methods for changing the position and the size of the created instance plus above mentioned methods returning author's ID or name.
- The library class for generating the assignments which offers a method that gets the student's ID as the parameter and then returns corresponding version of assignment.

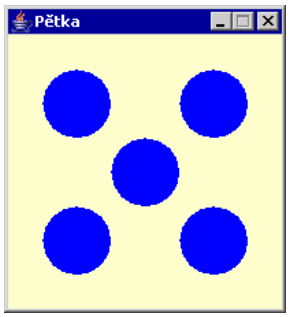

Figure 1: The Five

- The test class whose instances test that the instances of the created class correctly implements all methods declared in the assignment.

The library class is simple: its method finds out the hash code of the given ID and derives from it the initial position of the die and the shapes forming its "points". The differences among particular assignments are small, but for our purpose they are sufficient.

6.2 Calculator

As the second example I introduce my favourite calculator. The students' task is to create a part of a larger project – the class representing its CPU.

Their concrete task is to create a class implementing the interface `ICPU` that declares the requested behaviour or the created CPU.

The remaining parts of the project are prepared by the teacher. Here belongs the class `RealGUI` that takes care about the user interface, the class `TestCPU` for testing of the created CPUs and the class `Version` that generates assignments and test steps for testing this assignment.

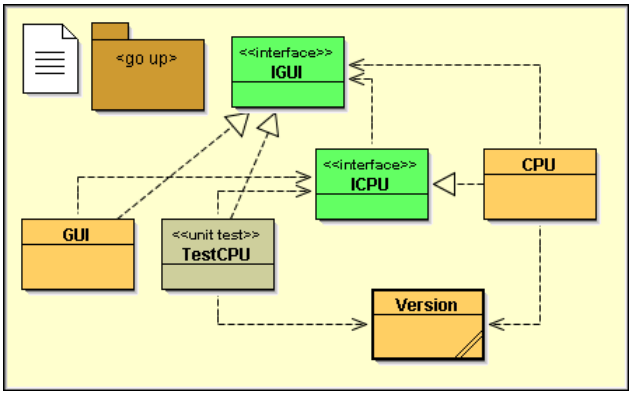

Figure 2: The Calculator

In this case the assignment is not generated only according to the student's ID. In addition to his/her id each student enter the level of difficulty of the generated assignment. Naturally solving of the assignment with higher level of difficulty yields also higher level of obtained points.

The class `Version` returns the assignment as a list of caption on buttons that the student's CPU has to serve. The student than programs the particular functions (that means responses to pressing of

particular buttons). In every moment of his/her development the student can create an instance of the class **`TestCPU`** and ask it what function of the developed class work and what not.

The instance of test class works in such a way, that it asks the student's class for the student's ID and level of difficulty. Than it asks the class **`Version`** for a corresponding assignments and list of test steps. Thereafter it connects to the student's class as a GUI, sends it a sequence of messages about pressing the calculator button and after each message gets the answer of the student's class and compare this answer with the requested one (it is a part of the test step).

After solving the assignments the students save their solutions into a specified folder with remaining classes belonging to the project. Then I run my "mass test" giving them the class object of the interface **`ICPU`** and an instance of a test class implementing the interface **`ITest`**. It checks all 60 assignments in one minute and save the protocol into the specified file.

Students, whose solutions passed the "mass test" may come to defend their solution. Here they get some small additional assignments how to modify their solution. When they finish the modification, they receive the appropriate number of points.

6.3 Conversation game

In the later lessons the students should create an application that implements a conversational game. In this game the player is going through a virtual world, asks the computer some questions and according its responses he decides how to continue. So 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.

The first rule that can help us in our attempt to automate evaluation of such solutions is to set some borders for all handed in solutions. I therefore defined a framework where the students should put their solutions (see fig. 3).

The class diagram in this case is a bit more complicated despite some of the dependences are not shown there. Such an assignments is not solvable with one class. Instead whole set of cooperating classes must be used.

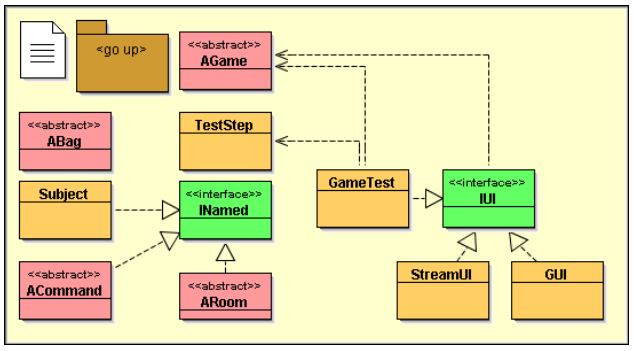

Figure 3: The project Game

If we want reasonably keep track of the received particular solutions, we need to ask students to put their solutions into special packages named by their ID. The class names inside theses packages are then not important.

So we have a framework defining borders and rules. We need to have a set of tests. However we cannot design these tests for all students because we don't know what game will each particular student design and program. The first part of the assignment is therefore to draft a scenario of their game. This scenario should be designed as a sequence of test steps simulating behaviour of a player. Every step should be an instance of the class **`TestStep`**. These instances should contain following information:

- The command entered by the user at the start of this step and the application has to react on it.
- The message with answer of the application on the entered command.
- The room (or its equivalent) where the user comes after processing this command by the application.
- The set of exits from the target room (that means the set of rooms where the player can go from the target room).
- The set of objects occurring in the target room.
- The actual content of the user's bag after processing the command.

Instances of the class **`GameTest`** have two methods:

- The method **`testTests(AGame)`** simulates behaviour of the game in such a way, that it goes through particular test steps and writes the requested game states after particular commands. This method is designed especially for teachers to be able to quickly evaluate the presented scenarios and estimate, if such a game is appropriate to the abilities of the particular student.
- After confirming the presented scenario the students start to develop the program. Their task is simple: develop a program behaving exactly in the way defined by the confirmed scenario. For testing this program serves the method **`testGame(AGame)`** that sends to the tested program the sequence of test steps and checks if the program responses in the way requested by the scenario.

The final test of the handed in solutions is again simple. The teacher copies all packages into one project and let the test library search all the classes extending the class **`AGame`**. Asks each found class for its author, opens the saved scenario and checks how the behaviour of this class (more precisely behaviour of the application represented by this class) corresponds to the behaviour requested by corresponding scenario.

7. CONCLUSION

The paper showed that the appropriate use of the interface gives in our hand very powerful tool for designing assignments and especially for following evaluation of the handed in solutions. It demonstrated how to prepare assignments forcing students to master the skills that they need in their future 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 evaluation of handed in programs. It use can significantly shorten the time needed for this evaluation.

The use of methodology *Design Patterns First* that put the explanation of the interface in the very beginning lessons allows to use all these advantages for the all time of the course.

8. REFERENCES

[1] BERGIN, Joseph: Fourteen Pedagogical Patterns. *Proceedings of Fifth European Conference on Pattern Languages of Programs.* (EuroPLoP™ 2000) Irsee 2000.

[2] BLOCH, Joshua. *Effective Java – Programming Language Guide*. Addison-Wesley Professional © 2001. 252 s. ISBN 0-201-31005-8. (Český překlad: *Java efektivně – 57 zásad softwarového experta*. Praha: Grada © 2002. 230 s. ISBN 80-247-0416-1)

[3] *Computing Curricula 2001, Computer Science Volume.* http://www.sigcse.org/cc2001/

[4] GAMMA, Erich; HELM, Richard; JOHNSON Ralph; VLISSIDES, John. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, © 1995. 396 s. (Český přelad: *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha: Grada, © 2003. 386 s., ISBN 80-247-0302-5)

[5] KERIEVSKY Joshua: *Refactoring to Patterns*, Addison Wesley, © 2005, 368 stran, ISBN 0-321-21335-1

[6] METSKER Steven John: *Design Pattersn Java Workbook.* Addison-Wesley, © 2002, 476 stran, ISBN 0-201-74397-3.

[7] 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.

[8] PECINOVSKÝ Rudolf: *Návrhové vzory*, Computer Press, © 2007, 528 stran. ISBN 978-80-251-1582-4.

[9] PECINOVSKÝ Rudolf: Aplikace metodiky „Design Patterns First“. *Objekty 2005 – sborník příspěvků devátého ročníku konference*, VŠB, Ostrava 2005. ISBN 80-213-1568-7.

[10] 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.

[11] PECINOVSKÝ Rudolf: Začlenění návrhových vzorů do výuky programování. *Objekty 2005 – sborník příspěvků devátého ročníku konference*, VŠB, Ostrava 2005. ISBN 80-248-0595-2.

[12] PECINOVSKÝ Rudolf: Jak efektivně učit OOP. *Tvorba softwaru 2005 – sborník přednášek*. ISBN 80-86840-14-X.

[13] PECINOVSKÝ Rudolf: *Myslíme objektově v jazyku Java 5.0*, Grada, © 2004, 602 stran, ISBN 80-247-0941-4.

[14] SCHMOLITZKY, A.: “Objects First, Interfaces Next” or Interfaces Before Inheritance. In *Proc. OOPSLA '04 (Companion: Educators' Symposium), (Vancouver, BC, Canada, 2004)*, ACM Press.

[15] SCHMOLITZKY, A.: Teaching Inheritance Concepts with Java. *Proceedings of International Conference on Principles and Practices of Programming In Java (PPPJ 2006)* Manheim, 2006

[16] Shalloway, A., Trott, J. A. *Design Patterns Explained – A new Perspective on Object-Oriented Design (2nd edition)*. Addison-Wesley, 2004. ISBN 0-321-24714-0.

[17] STELTING Stephen, MAASSEN Olaf: Applied Java Patterns, Sun Microsystems Press, © 2002, 576 stran, ISBN 0-13-093538-7

[18] *The ACM Java Task Force – Project Rationale*, Second Public Draft (February 23, 2006), ke stažení na adrese http://www-cs-faculty.stanford.edu/~eroberts//jtf/rationale/rationale.pdf