In What Order Should I Correct the Exercises?

Determining the Evaluation Order for the Automatic Assessment of Programming Exercises

Salvador España & David Insa & Josep Silva & Salvador Tamarit
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
E-46022 Valencia, Spain.
{sespana,dinsa,jsilva,stamarit}@dsic.upv.es

I. Introduction

The assessment of programming code is an arduous and time-consuming task. For instance, in university Java programming courses, many exercises and exams are projects, often composed of a set of classes that the student must complete and usually complement with new classes. In programming, there are often many different (usually infinite) correct solutions to the same problem, and this forces the teacher to inspect all delivered files, and to test the student's code to identify errors. It has been measured [5] that, in advanced courses, this process can be so complex that teachers make mistakes when correcting, with an average error over 5% of the mark.

Fortunately, new systems have emerged to overcome these problems. A comparison of the most important automatic assessment tools can be found in [4]. Other previous reviews are [2], [6], [1]. Most of the systems proposed are only limited to test case generation [8], [3], but others perform code analysis [9], [7] or use advanced reflective methods able to assess the code using the teacher's solution [5]. The most advanced systems of this kind allow the teacher to identify unsatisfied properties of the source code that are undetectable with test cases (e.g., an appropriate use of variable names). Unfortunately, the interference of different properties of the code can prevent these systems from completing the assessment.

Example 1.1: Consider the following Java code that should be implemented by a student:

```
class Car extends Vehicle          (A)
{
   int year;
   Car(int year)                   (B)
   {
     this.year = year;
   }
   int getPlateNumber()            (C)
   {
     return super.plateNumber;
   }
}
```

Being the assessment criteria: (A) 1 point = extending class `Vehicle`; (B) 1 point = constructor; (C) 1 point = method `getPlateNumber`. Clearly, C depends on A, but A does not depend on B or C, and B does not depend on A or C. This means that the assessment of this code should be done in one of the following topological orderings: ABC, BAC, ACB. Contrarily, the CAB, BCA, and CBA orders might produce compilation problems. For instance, if the student's code does not include the `extends` clause, then the `getPlateNumber` method cannot be compiled, even if it is correct. Hence, the system should correct first A, before trying to assess C.

Example 1.1 illustrates how the assessment order of programming code conditions the amount of code that is finally assessed. In an automatic assessment system, this order is extremely important, because the system cannot proceed if the order is wrong. In this work, we share our ideas about how to specify and implement the assessment order in the correction of programming exercises.

II. The Importance of the Assessment Order

The order chosen when assessing exercises highly influences the assessment process itself. Two different orders should be taken into account:

A) The order in which the exercises of an exam are assessed.
B) The order in which the exams (of different students) are assessed.

A. The Assessment Order of the Exercises

Defining a good order to process the exercises can benefit an assessment system, as it was shown in Example 1.1, but it can also be useful for the own teacher. In general, it is easier and more ergonomic (and thus quicker) for the teacher to inspect the exercises following an order that allows them to check, e.g., a function with the guarantee that the errors produced when executing this function are not caused by errors in other functions. This is illustrated in Figure 1, where a call graph is shown.

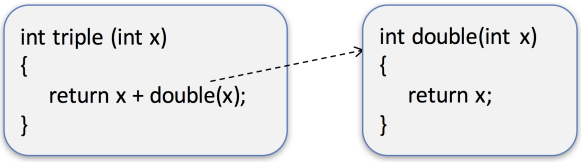

Fig. 1. Call graph showing an assessment dependency between two functions

Clearly, the execution of function `triple` depends on the execution of the (buggy) function `double` (because the former calls the later). Therefore, an error revealed when executing `triple` could be caused by either `double` or `triple`. For instance, `triple 5 = 10` reveals a bug caused by `double`. Thus, the teacher will have to assess the code of `triple` having in mind that the error could come from another function. Contrarily, if the assessment order starts in function `double`, only this function could be wrong when assessing it (because it does not depend on any other function). After `double` has been corrected, when assessing `triple`, the same good property will hold: The teacher can ignore the rest of the code and only focus on the code that is being examined.

We have implemented our ideas and integrated them into the automatic assessment system ASys. In the system, the user can specify properties that the code must fulfil. Each property has an associated mark, and it can also have dependencies with other properties. The dependencies form a (possibly unconnected) directed graph that we call Property Dependency Graph (PDG). For instance, the PDG associated with Example 1.1 would be formed by the A, B, and C vertices; and only one single arc $C \rightarrow A$. These dependencies are usually acyclic, although in real (more complex) examples there can exist direct or transitive cycles, which represent collections of properties that cannot be corrected isolatedly. In that case, the system is able to parallelize the correction process as far as dependencies are respected. Nevertheless, the supervision process should be done in a serial way in any topological order, some orders probably requiring less cognitive effort than others. Our implementation allows us to specify connected components (including cycles), and it determines the correction order in linear time.

Our implementation is publicly available at:

http://users.dsic.upv.es/~jsilva/ASys/

We used this tool in four real university programming courses (more than 400 students participated) with very satisfactory results:

- the correction process was able to correctly assess more code, and
- the time required to assess the exercises was reduced.

The relations among the exercises of an exam need to be specified. This means that our technique has a cost, because the teachers have to identify and specify these relations when preparing the exercise. However, the time required for this task is insignificant compared with the time it saves. Moreover, the more students solve the exercise, the more beneficial this technique is. The teachers that used the system with and without our technique valued our contribution as very useful.

B. The Assessment Order of the Exams

When a teacher has to assess a collection of programming exams, the only information they know a priori about the exams is the author. No information about the internal code is available before assessing it, and this makes difficult to establish a proper assessment order. In fact, the assessment order is usually alphabetical.

Nevertheless, we believe that, since the ordering of the exams to be corrected may influence the quality of the assessments and the teacher effort and comfort, it is worth trying to take profit of this feature: we propose to order the students' exams in such a way that those exams with no bugs are assessed first. Then, all the exams that have one single bug in the same method. Then all the exams that have one single bug in the same class. And so on. Also, if it is possible, similar exams should be placed contiguously. The benefits of this procedure are:

- the assessment is made easier and, probably, fairer because the same bugs are assessed in a row, thus having in mind the previous assessment (to avoid unfair comparisons in the marking),
- this also speeds up the assessment, because the part of the code being assessed has been often already assessed in the immediately previous exam,
- correcting similar exams increases the probability of detecting copies, and, finally
- this feature opens the door for further improvements: it would be possible, in a future extension of ASys, to detect similar code from different exams, which are not necessarily due to copies, in order to reuse the teacher's criteria. For the sake of security, the system would not make an assessment reusing information from a previous evaluation without querying the teacher.

Through the use of testing, ASys is able to detect those methods and classes that have bugs. As it was discussed in [5], automatically correcting all bugs is undecidable, and teacher assistance is often needed. However, before prompting the teacher with an exam to be corrected, ASys can detect the bugs in the code, as well as other properties such as the similarity of the proposed solutions, and sort the exams with some criteria as explained before. This feature is perfectly feasible and it is currently being implemented in ASys.

There is a second idea that we want to implement related to the assessment order of exercises: instead of assessing one exam at a time, we can assess one property, method, or class at a time (e.g., assessing one method in all the exams, then another method, and so on). This has one problem: we cannot know any mark until the very end because, at any intermediate point, all the exams are partially assessed, but none of them is finished.

However, this idea has a number of benefits: First, the assessment is more fair, because it is easier for the teacher to apply the same criteria if all the methods of one kind are assessed in a row. Moreover, the assessment of different exams is less affected by the mood of the teacher.[1] Second, the assessment is faster because, most of the time, the teacher has in mind the solution of the function being assessed.

In Figure 2 we see how to combine the ideas proposed. First,

[1]Take into account that the assessment of, say, 50 Java projects can last several days.

given an exam, it is possible (and automatic) to construct its associated PDG. Moreover, ASys can automatically detect for each exam the specific properties that fail. Thus, the assessment order can be automatically determined: all unsatisfied properties with no dependencies first (in the figure, all exams pointed by Property 1) and then, iteratively, all properties whose dependencies have been already assessed (in Figure 2, all exams pointed by Property 2 first, and so on).

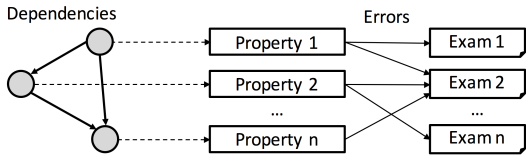

Fig. 2. Automatically determining the evaluation order

III. Conclusions

This work addresses the influence and the importance of the assessment order, both within exams and also among the exams from different students. This may basically determine the amount of code that is finally assessed, the teachers' productivity and ergonomics and, also, the accuracy of the marks. The scheme proposed is able to automatically determine an assessment order. For this, first, a graph of dependencies among the exercises that compose an exam is built. Second, the exercises of the exam are assessed following the dependencies, but not necessarily one exam at a time. The assessment could jump from one exam to another with the objective of assessing similar problems together, thus improving the efficiency and quality of the assessment.

IV. Acknowledgements

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic). It has been also supported by the *Escuela Técnica Superior de Ingeniería Informática* and by the *Instituto de Ciencias de la Educación* under grant PIME 2016-B18 at the *Universitat Politècnica de València*. Salvador Tamarit was partially supported by the *Conselleria de Educación, Investigación, Cultura y Deporte de la Generalitat Valenciana* under the grant APOSTD/2016/036.

References

[1] K. Abd Rahman and M. Jan Nordin. A review on the static analysis approach in the automated programming assessment systems. In *National Conference on Programming 07*, dec 2007.

[2] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, feb 2005.

[3] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. CodeWrite: Supporting student-driven practice of Java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 471–476, New York, NY, USA, 2011. ACM.

[4] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppala. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, New York, NY, USA, 2010. ACM.

[5] D. Insa and J. Silva. Semi-automatic assessment of unrestrained java code: A library, a dsl, and a workbench to assess exams and exercises. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2015, Vilnius, Lithuania, July 4-8, 2015*, pages 39–44, 2015.

[6] Y. Liang, Q. Liu, J. Xu, and D. Wang. The recent development of automated programming assessment. In *International Conference on Computational Intelligence and Software Engineering (CiSE 2009)*, pages 1–5. IEEE, dec 2009.

[7] K. A. Naudé, J. H. Greyling, and D. Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, feb 2010.

[8] F. Prados, I. Boada, J. Soler, and J. Poch. Automatic generation and correction of technical exercises. In *International Conference on Engineering and Computer Education (ICECE 2005)*, 2005.

[9] T. Wang, X. Su, Y. Wang, and P. Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107, feb 2007.