How to Apply Software Engineering Testing Methodologies to Education

Francisco Zapata[1], Olga Kosheleva[2], and Vladik Kreinovich[3]
[1]Department of Industrial, Manufacturing, and Systems Engineering
[2]Department of Teacher Education
[3]Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, Texas 79968, USA
fazg74@gmail.com, olgak@utep.edu, vladik@utep.edu

Abstract

Testing is a very important part of quality control in education. To decide how to best test, it makes sense to use experience of other areas where testing is important, where there is a large amount of experimental data comparing the efficiency of different testing strategies. One such area is software engineering. The experience of software engineering shows that the most efficient approach to testing is to test thoroughly on every single stage of the project. In regards to teaching, the resulting recommendation means making testing as frequent as possible, preferably giving weekly quizzes. At first glance, this may seem difficult, since grading quizzes – especially for big classes – requires a lot of time, and instructors usually do not have that much time. This problem can be solved by giving multiple-choice quizzes for which grading can be automatic. Automatic grading also helps make grading more objective – and thus, eliminate perceived grading subjectivity as a potential problem affecting student learning.

1 Formulation of the Problem

In education, adequate testing is important. When and how to test is an important issue in education, an issue that affects the teaching results.

On the one hand, we have only a limited time that we can allocate to testing. Within this limited time, it is not possible to ask the students in detail all about the parts of the material. And if some topic is not well-tested (or even not tested at all), students will not learn this topic well.

On the other hand, if we try to test everything and spend too much time on testing, this takes time out of teaching itself – this is, e.g., what many high school teachers are complaining about, that a lot of time is spent not so much on learning new material, but on testing and teaching how to prepare for the tests.

With all this importance, testing remains more art than science. There are no universal well-justified rules for how many tests and quizzes to give, how to grade them, etc. From this viewpoint, any well-justified recommendation can be helpful.

Similar problems exist in software engineering. In education, our goal is to make sure that students know the material and can correctly solve the corresponding problem. Testing is a main way of quality control, of checking that the students indeed acquired the corresponding knowledge and skills.

There is another area where testing is important – software engineering, whole goal is to make sure that the resulting software correctly performs the corresponding tasks; see, e.g., [1, 2, 3, 4, 8, 9, 10, 11]. There too, testing is a main way of quality control, of checking that the software indeed works correctly.

Of course, testing is important not only for software engineering, it is also very important for systems engineering in general, where we also need to test both the components of a system and the system itself; see, e.g., [5, 6, 7, 12].

Software engineering is where there is the largest amount of empirical data about different testing strategies. While testing is ubiquitous in many disciplines, software engineering has accumulated probably the largest amount of empirical data about different testing strategies. The reason for this relatively large amount of empirical data is that:

- in contrast to hardware whose testing is often complicated and expensive,
- testing software is relatively easy.

In a short period of time:

- we can easily test the given software on hundred and thousands of different inputs,
- while an equivalent number of hardware tests would require thousands of dollars and months of time.

An additional advantage of software testing versus, e.g., testing in education is that:

- In education, we cannot easily experiment on real students – we want them to learn, so we cannot test on students new risky teaching strategies that have a high probability of failure – such experiments would never be approved by the corresponding regulators who make sure that students are not hurt by such experimentation.
- In contrast, when testing software, we can try different strategies without any risk to hurt anyone – as long as after all this testing, we have a reasonable degree of confidence that the resulting software works correctly.

Resulting idea. Since software engineering has accumulate most data about different testing strategies, it is reasonable to apply the findings from software engineering to teaching.

2 Our Idea

Main conclusion of software engineering testing experience: from waterfall model to V-model. One of the main conclusions that researchers have made based on the experience of software testing is that we need to go from the original waterfall model to the newer V-model; see, e.g., [1, 2, 3, 4, 8, 9, 10, 11].

Crudely speaking, in a waterfall model, we go through several stages of designing software:

- eliciting specifications,
- designing a prototype, etc., all the way to
- the actual coding,
- after which we test the resulting code.

If some flaws are discovered during this testing (and usually, some flaws are discovered), we go back and repeat the cycle again and again until the resulting code satisfies all the tests.

The problem with the waterfall model is that some of the program errors are caused by the errors at early stages of the process:

- misunderstanding of not-exactly-precisely formulated specifications,
- mistakes in the prototype, etc.

These early-stage errors are not discovered until we test the code, and, as result, a lot of effort and resources is wasted on detailing a faulty design.

To rectify this situation, V-testing means serious testing on each stage, from the specification to prototyping to the actual coding.

How can we use this idea in teaching. Often, in classes, students are only seriously tested a few times:

- at a midterm exam (or at two or three midterm exams) and
- at the final exam.

This sparse testing corresponds to the waterfall model in software engineering, in the sense that initial misunderstandings are only checked and found out much later, in a month or so, during a test.

From this viewpoint, it would be more beneficial to test at every stage, i.e., to have comprehensive weekly quizzes in addition to midterm exam(s) and the final exam.

Resources are limited. The usual argument against weekly quizzes is that, especially in big classes, they require a lot of time to grade, and instructors do not have that much time. A solution is to use *automatic* testing tools, e.g., use multiple-choice quizzes and automatic systems for grading these quizzes.

At first glance, multiple-choice quizzes are more appropriate for simple subjects, but some of us (FZ) efficiently used such quizzes in teaching complex computer science subjects as well: e.g., a question on a quiz can include a complex piece of the code with embedded loops, and possible choices include different values computed by this code fragment.

Designing a multiple-choice quiz – especially for complex classes – takes time, but this is a time well spent: having the students take these quizzes helps them study – and gives the instructor a clear understanding of where each student stands, which, in its turn, helps the instructor take this into account and teach students the best possible way.

Objectivity: additional advantage of automatic testing. An additional advantage of automated testing is that the results of this testing are objective, there is no room for subjectivity, for how many points of partial credit to give for a partially solved problem. Thus, there is no possibility for arguments and for students being unhappy with the perceived bias and/or subjectivity.

Frequent quizzes help students learn – and help them pass tests. It is known that testing is not only gauging the student's level of knowledge, it enhances their knowledge – by having them to recall the material.

Also, a constant practice makes student get an experience in taking tests – which helps them take future tests.

Acknowledgments

This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science) and HRD-1242122 (Cyber-ShARE Center of Excellence).

References

[1] B. Biezer, *Software Testing Techniques*, Wiley, 2008.

[2] E. J. Braude and M. E. Bernstein, *Software Engineering: Modern Approaches*, Waveland Press, Long Grove, Illinois, 2016.

[3] S. Cha, R. N. Taylor, and K. Kang (eds.), *Handbook of Software Engineering*, Springer, Cham, Switzerland, 2019.

[4] M. Hoffman and T. Beaumont, *Application Development: Managing the Project Life Cycle*, Midrange Computing, Carlsbad, California, 1997.

[5] INCOSE, *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, Wiley, Hoboken, New Jersey, 2015.

[6] D. Liu, *Systems Engineering: Design Principles and Models*, CRC Press, Boca Raton, Florida, 2015.

[7] NASA, *NASA Systems Engineering Handbook: NASA/SP-2016-6105 Rev2*, NASA, Washington, DC, 2017.

[8] R. S. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 2015.

[9] V. Rajlich, *Software Engineering: The Current Practice*, Chapman & Hall/CRC, Boca Raton, Florida, 2016.

[10] I. Sommerville, *Software Engineering*, Pearson, Harlow, UK, 2018.

[11] F. Tsui, O. Karam, and B. Bernal, *Essentials of Software Engineering*, Jones and Batlett Learning, Burlington, Massachusetts, 2016.

[12] C. S. Wasson, *System Engineering Analysis, Design, and Development: Concepts, Principles, and Practices*, Wiley, Hoboken, New Jersey, 2015.