# Addressing Teaching Practices Regarding Software Quality: Testing and Debugging in the Classroom

Tilman Michaeli
Computing Education Research Group
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstraße 3, 91058 Erlangen, Germany
tilman.michaeli@fau.de

Ralf Romeike
Computing Education Research Group
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstraße 3, 91058 Erlangen, Germany
ralf.romeike@fau.de

## ABSTRACT

Software quality is seen as an integral part of CS education. Two of the key concepts concerning software quality are testing and debugging. Testing is considered important to verify the students' underlying model or algorithm. Debugging is an approach related to computational thinking which is distinct from general programming skills and fosters abilities like logical reasoning and independent problem solving. However, approaches, teaching materials, and studies on how to teach and integrate those concepts effectively into K12 classrooms are lacking. Therefore, both debugging and testing are often neglected in teaching practice, despite them being represented in many (but not all) curricula. In the following, we present a research project with the intention of providing adequate and evaluated strategies for addressing software quality in the classroom and its rationale. For this purpose, the model of Beizer's testing levels has been utilized and didactically transposed, thereby making it applicable to CS education in K12. The resulting categories may provide a basis for teaching and research.

## CCS CONCEPTS

• **Social and professional topics → K-12 education**;

## KEYWORDS

software quality, debugging, testing, teaching practice, CS education

**ACM Reference format:**
Tilman Michaeli and Ralf Romeike. 2017. Addressing Teaching Practices Regarding Software Quality: Testing and Debugging in the Classroom. In *Proceedings of WiPSCE '17, Nijmegen, Netherlands, November 8–10, 2017,* 2 pages.
https://doi.org/10.1145/3137065.3137087

## 1 INTRODUCTION

Students are confronted with software quality (SQ) or lack thereof every day: They experience bugs in the programs and software they employ on a regular basis, while hearing about security leaks in the media. Testing and debugging as methods for quality assurance are also important in CS education. By improving testing skills, learners produce higher quality code and learn to improve their program reasoning and abstract thinking skills [7]. A high level of debugging proficiency leads to independence when encountering problems, and to improved program comprehension and confidence. Eventually, learners may prefer a systematic and planned approach instead of trial-and-error [1]. Students as well as teachers often do not know the difference between testing and debugging [10]. Furthermore, experience shows that in classrooms, testing and debugging are the first things to be left out if time runs short, despite the presence of tool support (e.g. [11]). This is in stark contrast to the time and resources a professional software developer spends on both activities. Overall, there is a lack of approaches and materials for K12 that might help to resolve these problems in teaching practise. In the following, we present a research project addressing this disparity. This is done by identifying existing approaches for the integration of SQ, by providing a model of SQ understanding (derived from Beizer's testing levels), and by classifying the approaches into the model.


*WiPSCE '17, November 8–10, 2017, Nijmegen, Netherlands*

ACM ISBN 978-1-4503-5428-8/17/11.
https://doi.org/10.1145/3137065.3137087

## 2 RELATED WORK

In the software context, debugging is the process of finding and correcting errors in programs. In a broader sense, debugging is a computational thinking approach, which goes beyond fixing erroneous code: It involves applying logical reasoning and strategies to overcome real life problems. Debugging chronologically follows testing: By debugging, the developer corrects errors previously highlighted by testing. At the same time, testing can be seen as a debugging strategy [10]. Because of their common traits, an approach that unifies both concepts via the context SQ seems adequate and necessary.

We analyzed international and German curricula and found that SQ, testing, and debugging have been considered increasingly important in recent years (e.g. in the ACM K12 curricula recommendations 2011 vs. 2016): Every curiculum examined contained either testing, debugging, or both. However, there is a surprising lack of concrete approaches and materials.

For testing, there appears to be a consensus that it can not be taught as an isolated topic but should be incorporated throughout the whole curriculum (e.g. [5]). Existing research concerning the teaching of debugging implies that it should be taught explicitly (c.f. [4, 9]). There are existing approaches in literature (many of them on a university level), which can be assigned to four levels:

*(a) Debugging exercise:* The learners are confronted with a piece of source code containing a varying number of semantic or logical errors and have to fix them using debugging strategies (e.g. [4]).

*(b) Test-driven development:* For each program functionality, the students have to write the tests before its implementation (e.g. [6]).

*(c) Cross testing:* All learners write tests which are applied to their co-students' implementations (e.g. [8]).

*(d) Code review:* In pairs or groups, the learners read through the code, follow the control flow and therefore "execute" the program themselves, assuming the role of the computer (e.g. [2]).

Beizer's testing levels provide a categorization for the goals of professional testers [3]. He therefore identifies five levels, reflecting the maturity level of their test process. Over the course of their career, most software developers are likely to progress through these levels.

## 3 UNDERSTANDING SQ: 5 LEVELS

We developed a didactically transposed learning process version of Beizer's categorization which seems promising for teaching and research with respect to SQ.

**Level 0: Software that compiles works.** A level 0 student focuses on making a project compile and run, therefore seeing no difference between testing and debugging.

**Level 1: Software that successfully processes sample data works.** On level 1, the learner attempts to show that the software works by letting the program successfully run with some arbitrary or instructor-given sample data. This view is reported to be common for students as well [7].

**Level 2: By using edge cases one should show that the software does not work.** A student understands that the purpose of testing is to show that the software does not work, e.g. by building equivalence classes and thinking of edge and special cases ("If you test software thoroughly, it will work perfectly").

**Level 3: Testing and debugging can't prevent but reduce the risk of software failures and therefore improve SQ.** A learner on this level is aware of the phenomenon visible in everyday life: Testing and debugging may improve SQ but even professionals are not able to guarantee a software free of bugs. A residual risk of software failures remains. Testing and debugging are important for reducing the risk of such failures and preparations should be taken for analyzing issues found later.

**Level 4: Testing and debugging help with looking for mistakes and suggesting improvements in any scenario.** Learners on the highest level transfer their debugging and testing skills and the underlying principles from software development to general problem solving in the sense of computational thinking.

## 4 DISCUSSION

This model might help to assess the students' current level and thereby customize and adjust the teaching accordingly. Achieving at least a level 3 degree of understanding should be the goal for teaching practice, as levels 0 to 2 are contextualized to specific classroom tasks and activities. This may present an obstacle for the transfer and application of knowledge in a broader sense. But which levels of understanding are addressed with current approaches?

*(a) Debugging exercise: 1-2.* The students see that while the program compiles, it still has errors. Depending on the exercise, they locate those errors by using sample data or edge cases to investigate further. If they test properly, they eliminate all errors and the program works.

*(b) Test-driven development: 3-4.* Students might experience that despite their testing efforts, there are still errors in their programs. Using test-driven development, they use testing and debugging as methods in various contexts and for different functionalities. This helps foster the transfer to general problem solving.

*(c) Cross testing: 2.* The students try to show that their classmates' programs do not work properly by including edge and special cases.

*(d) Code review: 1-2.* The students go through the code step by step using a concrete problem instance, or special and edge cases to find errors. Due to the static nature of this approach, they may not notice their code might still be erroneous.

Many existing approaches may not enable learners to reach level 3, although they typically attempt to empower learners to generalize their testing and debugging skills for problem solving. This raises the question whether level 3 is required to advance to level 4 – or are students able to transfer their testing and debugging skills from lower levels to problem solving in other contexts? The levels might represent the steps in the learners' usual software development process: First, the program needs to compile (level 0). Subsequently, tests are conducted with sample data (level 1) and edge cases (level 2). Despite these efforts, the software might still contain errors (level 3). Thinking of this model in a strictly hierarchical sense makes it difficult to assign a position to the fourth level. Therefore, further research needs to determine if the fourth level is part of this hierarchy or separate? Regardless of the structure of the model, the most important question appears to be how to reach the third/fourth level.

The next step in the research project is to develop a assessment tool for the current level of a learner. Afterwards, existing approaches will be further analyzed and new approaches will be developed. These results will be empirically validated with respect to our model to provide materials and solutions and therefore address their lack in the teaching practice.

## REFERENCES


[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice Computer Science students. *ITiCSE '05* 37, 3 (2005), 84–88.

[2] David J Barnes and M Kölling. *Objects First with Java: A Practical Introduction Using BlueJ, (2006).* Prentice Hall.

[3] Boris Beizer. 1990. *Software Testing Techniques* (2 ed.). Van Nostrand Reinhold Co., New York, NY, USA.

[4] Ryan Chmiel and Michael C Loui. 2004. Debugging: from Novice to Expert. *SIGCSE '04* 36, 1 (2004), 17.

[5] Henrik Bærbak Christensen. 2003. Systematic testing should not be a topic in the computer science curriculum! *ACM SIGCSE Bulletin* 35, 3 (2003), 7.

[6] Chetan Desai, David S. Janzen, and John Clements. 2009. Implications of integrating test-driven development into CS1/CS2 curricula. *ACM SIGCSE Bulletin* 41, 1 (2009), 148.

[7] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. *ACM SIGCSE Bulletin* 36, 1 (2004), 26.

[8] Michael H. Goldwasser. 2002. A gimmick to integrate software testing throughout the curriculum. *ACM SIGCSE Bulletin* 34, 1 (2002), 271.

[9] Irvin R. Katz and John R. Anderson. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human-Computer Interaction* 3, 4 (1987), 351–399.

[10] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. *SIGCSE '08* 40 (2008), 163.

[11] A. Patterson, M. Kölling, and J. Rosenberg. 2003. Introducing unit testing with BlueJ. *ITiCSE '03* June 2003 (2003), 11–15.