Challenges to integrate software testing into introductory programming courses

Lilian Passos Scatalon, Ellen Francine Barbosa
University of Sao Paulo (ICMC–USP)
Sao Carlos–SP, Brazil 13560–970
Email: lilian.scatalon@usp.br, francine@icmc.usp.br

Rogerio Eduardo Garcia
Sao Paulo State University (FCT–UNESP)
Presidente Prudente–SP, Brazil 19060–900
Email: rogerio@fct.unesp.br

Abstract*—Several studies suggest that the teaching of software testing should begin as early as possible, since introductory programming courses. In this way, the teaching of both subjects, programming and testing, becomes an integrated teaching approach. Testing practices in this context can provide a timely feedback to students while they are still working on programming assignments and, as a result, increase the quality of their code. Besides, developing students' testing skills earlier is useful to improve their programming skills as well, since both kinds of skills are complementary. However, this integration is not straightforward, because lecture hours and the coverage of programming topics must remain the same, while testing concepts and practices are introduced. For this reason, when designing an educational approach to introduce testing practices to novice programmers, there is also the need to address potential difficulties faced by both students and instructors. Considering this scenario, this paper aims at identifying the challenges raised due to the integration of software testing into introductory programming courses. The goal is to provide support for instructors who intend to adopt the integrated approach. The challenges have been identified from the results of a systematic mapping we conducted of the literature in this domain. The main contribution of this paper refers to the establishment of a catalog of challenges faced to integrate software testing into introductory programming courses. We also discuss possible solutions to design courses using the integrated approach and point out challenges that have been scarcely addressed in the literature. Finally, we indicate directions that can be explored in future educational empirical studies in this context.

I. Introduction

Despite software testing being very important in industry, students are majoring in computing programs with deficiencies in their testing skills [21, 66]. This issue leads to rethinking how software testing education is done in the Computer Science curriculum.

Traditionally, software testing has been taught in upper level computing courses, such as Software Engineering or even elective courses on this subject [23]. However, by using this approach, students are missing the opportunity to apply software testing in their own programs written across previous computing courses. Also, they are not being able to further develop testing habits and skills [55].

There is a lot of effort in teaching students how to write code in computing courses. However, there is not as much effort in teaching them how to validate their own code, when in fact this could lead them to think more critically while writing the code. Therefore, in order to address this issue, the computing curriculum should give support to develop students' testing skills in parallel with their programming skills.

In this direction, it is widely recognized that, instead of being an isolated topic, software testing should permeate all computing curriculum in practical programming activities, aiming to emphasize the importance of producing high quality software [23, 55]. This curricular restructuring would imply teaching software testing earlier, beginning from introductory programming courses.

Firstly, factors related to how programming courses are designed should be considered to determine how this integration should be done. For instance, the programming language and platform chosen to present programming concepts to students define the supporting tools that can be used to conduct testing practices.

Nevertheless, the focus of introductory courses is to teach programming and it is not simple to add a new subject in already packed courses [40]. The integration of software testing into this context should be designed aiming to aggregate value, in such a way that it does not jeopardize the content and the flow of programming courses [56]. Instructors face several challenges to design and conduct such courses with testing practices. There are also challenges faced by students to keep up with this integrated approach.

Considering this scenario, in this paper we present a catalog of challenges raised by the integration of software testing into introductory programming courses. We intend to help instructors in the designing of such courses so that students can take as much benefits as possible from this integrated approach in terms of learning both programming and testing. The challenges were identified from the results of a systematic mapping we conducted in this domain.

The remainder of the paper is organized as follows. Section II points out aspects that should be considered to design the introductory sequence of programming courses. Section III presents the systematic mapping we conducted and whose results allowed to identify several difficulties reported by other researchers/instructors. In Section IV we describe these identified difficulties in the format of challenges that must be considered to design programming courses with software testing. Finally, Section V summarizes conclusions and future work directions.

978-1-5090-5920-1/17/$31.00 ©2017 IEEE

II. Background

Programming skills are required in many technology areas and, therefore, courses on this subject compose several undergraduate programs, especially those related to computing, such as Computer Science, Computer Engineering and Information Systems [2, 45]. Given the relevance of this subject, there is the need to explore different ways to design introductory courses with the goal of providing a learning environment as effective as possible to students.

The introductory sequence of computing courses involves the teaching of fundamental programming concepts that are supposed to support other following computing courses. However, there is not a consensus of how many courses should compose this sequence, neither of how the content should be distributed over them [16, 46, 70].

ACM/IEEE curricular guidelines also point out this existing diversity in the introductory sequence and identify some aspects that can vary in the design of these courses [4]:

- **Context:** The introductory courses differ greatly among institutions. An important aspect to consider is whether students are computing majors or not, since their needs as well as their motivation to learn programming may vary.
- **Programming focus:** The ultimate goal of introductory courses is that students learn basic computing concepts, such as abstraction and decomposition. In general, these concepts are taught by means of a programming language and the construction of programs. However, these general concepts can be taught without being tied to learning a programming language syntax.
- **Paradigm and programming language:** The programming paradigm choice is a decisive factor in the design of a introductory course, since it can influence greatly on the sequence that concepts are taught. Also, this choice can determine the whole underlying model of the introductory sequence (*imperative-first*, *objects-first* and *functional-first* [3]). Naturally, the choice of programming language is also related to the chosen paradigm. There are other important factors, such as language popularity, industry adoption (such as C, C++ and Java) or the simplicity of the syntax (like Python) [28].
- **Software development practices:** Considering the larger context of the software development process, programming is just one of its composing activities. In this sense, it is possible to include development practices that support programming, such as unit testing, refactoring and version control. The inclusion of such practices can help students in programming assignments and improve their notion of the development process.
- **Parallel processing:** The shift in computer hardware to multi-core processors has been influencing changes in Computer Science Education. There are initiatives to introduce notions of concurrency even in introductory courses [17, 44]. Still, it is more common that this subject is postponed to more advanced courses, given its difficulty.
- **Platform:** The diversity of platforms adopted during introductory courses has grown beyond traditional computers. For instance, there are initiatives to teach programming using mobile devices and robots [26, 34, 60]. The use of these alternative platforms can increase students' motivation, and, depending on the needs of the target audience, it can be very helpful. On the other hand, it is important to analyze if the programming concepts learned by means of such platforms are sufficient to establish the foundation for other advanced computing courses.

Instructors need to evaluate the tradeoffs involved in each one of these aspects to determine an appropriate design of introductory programming course for a given context [4]. There is not a "silver bullet", but being aware of these aspects and their implications while making design course choices can improve significantly the teaching of programming [83].

In particular, the use of testing practices stand out in the context of Computer Science introductory courses. Software testing is related to the aforementioned aspect of software development practices. Comparing ACM/IEEE curricular guidelines from 2001 and 2013 [3, 4], it is possible to notice the increase in the adoption of Software Engineering practices for novice programmers.

Furthermore, the teaching of software testing in this context not only is a supporting practice, but also composes the recommended content for the introductory sequence of computing courses, still according to ACM/IEEE curricular guidelines. More specifically, the recommended topics are *testing fundamentals and test case generation* and *unit testing* [4]. These topics should be taught aiming to provide to students a basic notion of how to validate their own code.

III. Systematic Mapping

Systematic mapping is a well-defined method to perform a literature review about a given research topic [63]. This kind of study supports gathering information from the literature in an organized and repeatable way, allowing to form a non-biased overview of the investigated topic.

We conducted this systematic mapping based on the guidelines from Petersen et al. [63]. Firstly, we defined the protocol, which is the plan to conduct the review and establishes the scope of the investigated research topic. The protocol is composed by research questions, the search strategy and the criteria to select relevant studies among the returned ones.

The goal of this study is to review the literature about software testing in the teaching of programming fundamentals. More specifically, the study was directed towards answering the research question:

- *What are the challenges faced to integrate testing practices into introductory programming courses?*

Next, we performed an automatic search in the following databases: ACM Digital Library[1], IEEE Xplore[2], Springer

[1]http://dl.acm.org
[2]http://ieeexplore.ieee.org

Link[3], Scopus[4] and Science Direct[5]. We selected these databases because they are among the most used ones by previous systematic literature reviews [87].

The search string was defined considering that the investigated research theme can be defined as the intersection of three aspects: programming, testing and teaching/learning. So, considering these aspects and after piloting, analyzing and refining it, we organized representative terms from this research theme as follows:

("*programming*" **OR** "*program*" **OR** "*code*" **OR** "*software*")
AND
("*testing*" **OR** "*test*")
AND
("*student*" **OR** "*course*" **OR** "*teaching*" **OR** "*learning*")

The **inclusion criterion** reflects the scope established by the research question: included papers should discuss or investigate software testing in the context of teaching programming fundamentals in higher education. The **exclusion criteria** were the following: duplicated studies, papers not written in English, research about software testing outside the context of higher education or that only address software testing in advanced computing courses, such as Software Engineering.

The search returned 16496 papers in total, 158 of which were selected. The low percentage of selected papers is due to the difficulty we had to determine more specific terms that aggregate more precision to the string. The selected studies were analyzed with the objective of determining the answer to the research question, which is presented in the next section.

IV. Challenges

The analysis of the selected studies from the systematic mapping allowed to identify difficulties and concerns reported in the literature. These identified elements expose the experience and evidence obtained by instructors/researchers that investigated and/or applied the integration of software testing into introductory courses. Next, we organized the results of this analysis in the format of challenges that instructors face to design programming courses with the integration of software testing.

A. Determine how and when software testing should be introduced in the curriculum

The first challenge refers to determine how the subjects of programming and testing should be connected and delivered in this context. This connection is not straightforward since programming courses are already packed [33, 40]. Jones [55, 56] suggests that this integration should be done in a holistic approach, with different kinds of testing practices that increase in terms of difficulty as students progress in the introductory courses.

If this integration is not carefully analyzed, the addition of software testing can greatly increase the effort required by both students and instructors. With regard to students, there is the risk of leading them to a cognitive overload. Students can learn only so many things at the same time, and the design of programming courses should ensure that they are being able to follow and apply the proposed practices. Otherwise, they would not be able to perceive software testing as a helpful practice (see Section IV-B).

Concerning the additional effort required by instructors, there may be a large increase in terms of work overload. Firstly, there is extra effort to prepare course materials and assignments [31, 85]. Assignments must be testable and, if test cases are provided to students, instructors also need to prepare those too. On the other hand, if students are required to write test cases, instructors have the additional responsibility of assessing students' tests. Additionally, instructors would have to provide feedback somehow for students, if possible, while they are still working on programming assignments. In general, the use of supporting tools can help considerably to reduce instructors' workload (see Section IV-F).

B. Help students appreciate the value of software testing

There is a widespread agreement that novice programmers are reluctant to conduct software testing [18, 20, 47]. When it is not made compulsory by the instructor, students tend not to do it [11]. Usually, to force students to conduct testing, part of their grade depends on it. However, this imposition can be increasing even more students' resistance, because while they do not really appreciate the value of testing, they can continue see it only as an unnecessary burden [30].

Students should be able to perceive in practice the benefits of testing their own programs. However, in general, projects from introductory courses (such as CS1 and CS2) are too simple to justify the additional effort of creating and executing thorough tests [5]. Students tend to superficially test their programs. After the submission they are often surprised to receive a grade that they do not expect, since their superficial testing went well [21]. This kind of late feedback probably will not help students' to improve their code, since they already finished working on the code.

Overall, there is the need of ensuring quality in programming assignments [24, 32, 85], so students feel the need to conduct software testing. In this direction, an appropriate configuration of testing activity to be applied during programming assignments (Section IV-C), timely feedback (Section IV-D) and the use of supporting tools (Section IV-F) can help to design an approach that helps students to indeed benefit from software testing in this context.

C. Determine how the testing activity should be conducted in programming assignments

When considering the binding of programming and testing activities, one can notice the widespread adoption of TDD (test-driven development) [30]. It consists in one of the key practices of the agile development methodology eXtremme Programming. TDD is a recurrent approach in introductory programming courses and its influence can be recognized

[3]http://link.springer.com
[4]http://www.scopus.com
[5]http://www.sciencedirect.com

in several other proposals of educational approaches in this context [9, 22, 30, 50, 52, 61, 62, 65, 67].

In TDD, there is a well-defined order for the activities of programming and testing. At first, the programmer develops test cases. This aspect is known as test-first programming. Next, he/she executes the test cases and finally writes the code to the ones that failed and apply refactoring as needed. This sequence of steps is repeated iteratively until the code unit is complete.

Still considering the order in which programming and testing activities should happen, the choice between test-first and test-last approaches is another important decision. According to the studies of Janzen and Saiedian [50, 53], early programmers are very reluctant to adopt a test-first approach, but their willingness increases as they move forward in programming courses.

From a general perspective, when considering the testing activity as a separate process, it can be composed by the following steps [29]:

- **planning**, where the testing strategy and needed resources are defined;
- **test cases design**, that consists in writing test cases, ideally based in established testing techniques and criteria;
- **program execution**, run the code along with the test cases; and
- **result analysis**, which consists in evaluating the results of test cases execution.

Instructors can ease this process considerably for students, so they do not need to be actively involved in all these steps. In the selected studies, students performed only the last three steps of the testing activity at most (test cases design, program execution and result analysis). It is implied in all papers that the instructor was responsible for the first step (planning the test activity).

The step of test case design brings an important issue: if students are going to be responsible for writing test cases or not. Test cases can be either required as a deliverable together with the solution code (student-written test cases) or provided to students as an scaffolding to develop the solution code (instructor-written test cases). Ultimately, instructors need to find a good balance between provided tests and tests students should write themselves [49].

Many factors can influence in this decision, such as instructor workload and students cognitive load, as discussed in Section IV-A. Nevertheless, it is important to consider that there may be a learning curve to learn how to write test cases [49].

The instructor conduct the test planning deciding which remaining steps students should be responsible for and how they will be conducted. Aiming to equip students to conduct the test activity, instructors can teach them basic testing concepts (Section IV-E) and stipulate the use supporting tools (Section IV-F). Moreover, students should be able to get feedback about their programming performance through the testing activity (Section IV-D).

D. Provide timely and useful feedback

In a typical programming assignment, students receive feedback only on the end result, preventing them from improving the quality of their solution [33]. Instead, students should receive constant feedback while working on their solutions in order to have the opportunity to improve their performance and learn from their mistakes [10, 33].

This challenge is related to the kind of assessment students will be subject to. Providing timely feedback is more related to formative assessment, where students will be able to improve their learning. If they only receive feedback after submitting the code, the assessment model gets more closer to summative assessment, and then students can miss the connection between thorough testing and a good programming performance or even the importance of developing high quality code [15, 43].

Therefore, the kind of feedback can also contribute to affect students' perceptions towards software testing, as discussed in Section IV-B. There are also studies that investigate how feedback can change students' testing behavior [19, 20].

Besides determining when and how often feedback is provided to students, other important aspect is the content of the feedback. Ideally, they should receive feedback on their performance in programming and, when they are supposed to write test cases, also in their testing performance.

The programming performance usually is calculated based on the *pass/fail rate* from test cases (either written by instructor or student). However, there is also the need to evaluate and provide feedback about students' test cases. More importantly, students should use this kind of feedback to improve their test suite (see Section IV-E).

E. Help students to become better testers

An analysis of students' test suites in [37] revealed that they were writing test cases to cover only common behavior rather than actually seeking to uncover errors on the solution code. The authors called this behavior as "happy path testing". These results show that, if students are supposed to write test cases, they should be instructed in how to write and improve a set of test cases, aiming to appropriately test a given program.

Aiming to help students improve their testing performance, we have identified the following important aspects to be considered: how their test cases should be assessed, whether students should receive instruction on fundamental testing concepts and what kind of feedback about their testing performance should be provided.

The most common way to assess quality in students' tests is through *code coverage*, that is, the percentage of code that has been exercised while running the tests. However, code coverage does not capture how much of expected behavior the test cases check.

Other metrics to assess test cases have been investigated, such as the ones based on *all-pairs testing* and *mutation analysis* [1, 38]. These metrics are better predictors of test quality than code coverage [36, 73], but they are more computationally expensive to obtain and raise some practical obstacles, mainly related to scalability. There are also proposals towards dealing with such obstacles [71, 72].

Although most studies adopt an approach with student-written test cases, it is rare to observe instruction on how to select values for test cases, with testing techniques and criteria. There are a few exceptions, like the studies in [6, 9, 24, 25, 40, 41, 57, 80, 86], where students were instructed on fundamental testing concepts. Without being aware of testing concepts, students are not sufficiently equipped to improve their testing performance [21].

On the same direction, the feedback provided to students about their testing performance could also be tailored based on testing concepts [21]. Feedback based on testing coverage is useful, but it does not guarantee that students are actually learning how to test better their programs. Besides, it can mean that they are just relying on trial-and-error, dependent of the test coverage tool to indicate pieces of code that were not covered yet.

Students need to understand underlying testing concepts to make the most of feedback based on obtained code coverage [76]. This kind of feedback can be more useful to students (see Section IV-D) and can help them to perceive purposefulness in the testing activity (see Section IV-B).

F. Choose appropriate supporting tools

Supporting tools ease the integration of software testing into introductory courses. Tools can help both students, in terms of motivation and cognitive load, and instructors, in terms of workload (see Section IV-A).

There are several types of tools, which automate different aspects of testing practices. We have identified the following types: *automated assessment tools* [35, 64, 75, 77, 84], *online judges* [42, 69], *games* [12, 13, 81] and *tutoring systems* [40, 51].

There are also testing frameworks/libraries, which consist in supporting mechanisms to write and execute test cases. The most used one is JUnit, even in the educational context [30]. However, frameworks and libraries like JUnit are not specifically designed for novice programmers, and there is the risk that their use can insert more difficulties into the learning of programming, instead of reinforcing it.

So, it is possible to observe testing libraries developed specifically to ease the learning curve for novice programmers, such as in [8, 14, 74, 78, 79]. There are also libraries designed to ease novice testing for concurrent programming and in other platforms [7, 68].

Using testing frameworks to implement test cases allows to perform automated testing of students' programs. In general, automated assessment systems run student code against test cases, which can be student and/or instructor-written. There are some proposals towards automating also the *test data generation* in these systems [48, 58].

An important detail that should be checked is the *interface conformance* between the solution code and the test cases, so they can be linked properly [54, 82]. Additionally, some precautions can be taken for the execution, such as running student code isolated in a *sandbox* to ensure security [59] and having mechanisms to *identify infinite loops* [39].

V. Conclusions and Future Work

In this paper we presented a catalog of challenges to integrate software testing in the context of introductory programming courses. This integrated approach can bring many benefits and help students to further improve their programming skills, but it also raises difficulties faced by both instructors and students. We aimed at presenting these difficulties from instructors' point of view, since they are responsible for decisions on how programming courses should be designed.

The goals of this integrated approach can be seen from two perspectives. From the perspective of teaching software testing, the idea is to teach this subject earlier and develop students' testing skills since the beginning of the Computer Science curriculum [27]. On the other hand, from the perspective of the teaching of programming, the idea is to apply testing practices to reinforce programming skills that would, otherwise, be left aside, such as comprehension and analysis skills [33].

So, in general, there is the need to consider the challenge of reconciling the goals from the teaching of both programming and software testing. Students' testing skills should be developed progressively over the introductory sequence and, at the same time, the introduction of software testing cannot disrupt programming courses' flow.

Students present a different level of programming skills in each programming course. Accordingly, it is possible to design testing practices of different levels of difficulty [56]. Therefore, there is the need to determine what kind of testing practice would be appropriate for each specific context of programming course.

The first step in this direction could be to address the challenges reported in this paper, since they summarize well-known difficulties to deliver this integrated teaching approach. Moreover, the conduction of empirical studies can help gathering evidence to investigate approaches configured according to different choices of course design. Ultimately, there is the need to understand how to achieve the best possible results for learning effectiveness of programming and testing.

In particular, we noticed some aspects that should be further investigated in the literature. The most widespread reported difficulty in the studies was the reluctance of students to conduct software testing, even when they recognize the importance of this practice in solving programming assignments. In parallel, the most popular configuration of testing activity is the TDD.

This combination arises the question if TDD is indeed the most adequate configuration to be adopted in every introductory programming course as it has been advocated [32]. Some studies report interesting results and discussions in this direction [5, 53], but it is still necessary to investigate the effect of isolated factors and help instructors in their design choices.

Other interesting aspect is the lack of instruction on fundamental testing concepts in this context [21]. It seems that the idea of adopting testing practices earlier in the curriculum is widely adopted, but the teaching of testing concepts, that would equip students to conduct the testing practices, is not. We are currently working in empirical studies to investigate both of these aspects.

As future work, we intend to investigate separate factors that influence this integrated approach, such as different configurations of the testing activity, testing concepts that must be taught, supporting tools that can be used etc. Also, we intend to investigate which results can be observed in order to assess learning effectiveness of programming and testing in this context.

Acknowledgment

This work is supported by FAPESP (Sao Paulo Research Foundation) grants 2014/06656-8 and 2016/17575-4.

References

[1] K. Aaltonen, P. Ihantola, and O. Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*, pages 153–160, New York, NY, USA, 2010. ACM.

[2] ACM/AIS/IEEE-CS. Computing curricula 2005, 2005. Joint Task Force on Computing Curricula, available at http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf.

[3] ACM/IEEE-CS. Computing curricula 2001, 2001. Joint Task Force on Computing Curricula, available at www.acm.org/education/curric_vols/cc2001.pdf.

[4] ACM/IEEE-CS. Computer science curricula 2013, December 2013. Joint Task Force on Computing Curricula, available at www.acm.org/education/CS2013-final-report.pdf.

[5] J. Adams. Test-driven data structures: Revitalizing cs2. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)*, pages 143–147, New York, NY, USA, 2009. ACM.

[6] R. Agarwal, S. H. Edwards, and M. A. Pérez-Quiñones. Designing an adaptive learning module to teach software testing. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*, pages 259–263, New York, NY, USA, 2006. ACM.

[7] A. Allevato and S. H. Edwards. Robolift: Engaging cs2 students with testable, automatically evaluated android applications. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, pages 547–552, New York, NY, USA, 2012. ACM.

[8] S. K. Andrianoff, D. B. Levine, S. D. Gewand, and G. A. Heissenberger. A testing-based framework for programming contests. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '03)*, pages 94–98, New York, NY, USA, 2003. ACM.

[9] E. Barbosa, J. Maldonado, R. LeBlanc, and M. Guzdial. Introducing testing practices into objects and design course. In *16th Conference on Software Engineering Education and Training (CSEE&T)*, pages 279–286, March 2003.

[10] E. Barbosa, M. Silva, C. Corte, and J. Maldonado. Integrated teaching of programming foundations and software testing. In *Annual Frontiers in Education Conference (FIE)*, pages S1H–5–S1H–10, Oct 2008.

[11] E. G. Barriocanal, M.-A. S. Urbán, I. A. Cuevas, and P. D. Pérez. An experience in integrating automated unit testing practices in an introductory programming course. *SIGCSE Bulletin*, 34(4):125–128, Dec. 2002.

[12] J. Bell, S. Sheth, and G. Kaiser. Secret ninja testing with halo software engineering. In *Proceedings of the 4th International Workshop on Social Software Engineering (SSE '11)*, pages 43–47, New York, NY, USA, 2011. ACM.

[13] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux. Code hunt: Experience with coding contests at scale. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*, pages 398–407, Piscataway, NJ, USA, 2015. IEEE Press.

[14] D. Blaheta. Unci: A c++-based unit-testing framework for intro students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*, pages 475–480, New York, NY, USA, 2015. ACM.

[15] M. K. Bradshaw. Ante up: A framework to strengthen student-based testing of assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*, pages 488–493, New York, NY, USA, 2015. ACM.

[16] K. B. Bruce. Controversy on how to teach cs 1: A discussion on the sigcse-members mailing list. *SIGCSE Bulletin*, 37(2):111–117, June 2005.

[17] K. B. Bruce, A. Danyluk, and T. Murtagh. Introducing concurrency in cs 1. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*, pages 224–228, New York, NY, USA, 2010. ACM.

[18] K. Buffardi and S. H. Edwards. Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*, pages 105–110, New York, NY, USA, 2012. ACM.

[19] K. Buffardi and S. H. Edwards. Impacts of adaptive feedback on teaching test-driven development. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, pages 293–298, New York, NY, USA, 2013. ACM.

[20] K. Buffardi and S. H. Edwards. A formative study of

influences on student testing behaviors. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*, pages 597–602, New York, NY, USA, 2014. ACM.

[21] J. C. Carver and N. A. Kraft. Evaluating the testing ability of senior-level computer science students. In *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 169–178, May 2011.

[22] M. E. Caspersen and M. Kolling. Stream: A first programming process. *ACM Transactions on Computing Education*, 9(1):4:1–4:29, Mar. 2009.

[23] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03)*, pages 7–10, New York, NY, USA, 2003. ACM.

[24] H. B. Christensen. Experiences with a focus on testing in teaching. In J. Bennedsen, M. Caspersen, and M. KÃ¶lling, editors, *Reflections on the Teaching of Programming*, volume 4821 of *Lecture Notes in Computer Science*, pages 147–165. Springer Berlin Heidelberg, 2008.

[25] J. Collofello and K. Vehathiri. An environment for training computer science students on software testing. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, pages T3E–6, Oct 2005.

[26] D. Cowden, A. O'Neill, E. Opavsky, D. Ustek, and H. M. Walker. A c-based introductory course using robots. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, pages 27–32, New York, NY, USA, 2012. ACM.

[27] T. Cowling. Stages in teaching software testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 1185–1194, Piscataway, NJ, USA, 2012. IEEE Press.

[28] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*, pages 625–630, New York, NY, USA, 2011. ACM.

[29] M. E. Delamaro, J. C. Maldonado, and M. Jino. *Introducao ao teste de software (Introduction to software testing)*. Elsevier, 2007.

[30] C. Desai, D. Janzen, and K. Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bulletin*, 40(2):97–101, June 2008.

[31] C. Desai, D. S. Janzen, and J. Clements. Implications of integrating test-driven development into cs1/cs2 curricula. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)*, pages 148–152, New York, NY, USA, 2009. ACM.

[32] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 148–155, New York, NY, USA, 2003. ACM.

[33] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*, pages 26–30, New York, NY, USA, 2004. ACM.

[34] S. H. Edwards and A. Allevato. Sofia: The simple open framework for inventive android applications. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*, pages 321–321, New York, NY, USA, 2013. ACM.

[35] S. H. Edwards and M. A. Perez-Quinones. Web-cat: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, pages 328–328. ACM, 2008.

[36] S. H. Edwards and Z. Shams. Comparing test quality measures for assessing student-written tests. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*, pages 354–363, New York, NY, USA, 2014. ACM.

[37] S. H. Edwards and Z. Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*, pages 171–176, New York, NY, USA, 2014. ACM.

[38] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil. Running students' software tests against each others' code: New life for an old "gimmick". In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, pages 221–226, New York, NY, USA, 2012. ACM.

[39] S. H. Edwards, Z. Shams, and C. Estep. Adaptively identifying non-terminating code when testing student programs. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*, pages 15–20, New York, NY, USA, 2014. ACM.

[40] S. Elbaum, S. Person, J. Dokulil, and M. Jorde. Bug hunt: Making early software testing lessons engaging and affordable. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 688–697, Washington, DC, USA, 2007. IEEE Computer Society.

[41] S. Frezza. Integrating testing and design methods for undergraduates: teaching software testing in the context of software design. In *32nd Annual Frontiers in Education (FIE)*, volume 3, pages S1G–1–S1G–4 vol.3, Nov 2002.

[42] O. Giménez, J. Petit, and S. Roura. Jutge.org: An educational programming judge. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)*, pages 445–450, New York, NY, USA, 2012. ACM.

[43] L. Gonzalez-Guerra and A. Leal-Flores. Tutoring model to guide students in programming courses to create complete and correct solutions. In *9th International Conference on Computer Science Education (ICCSE)*, pages 75–80, Aug 2014.

[44] T. R. Gross. Breadth in depth: A 1st year introduction to parallel programming. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*, pages 435–440, New York, NY, USA, 2011. ACM.
[45] M. Guzdial. Education: Teaching computing to everyone. *Communications of the ACM*, 52(5):31–33, May 2009.
[46] M. Hertz. What do "cs1" and "cs2" mean?: Investigating differences in the early courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*, pages 199–203, New York, NY, USA, 2010. ACM.
[47] M. Hilton and D. S. Janzen. On teaching arrays with test-driven learning in webide. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*, pages 93–98, New York, NY, USA, 2012. ACM.
[48] P. Ihantola. Test data generation for programming exercises with symbolic execution in java pathfinder. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, pages 87–94, New York, NY, USA, 2006. ACM.
[49] V. Isomöttönen and V. Lappalainen. Csi with games and an emphasis on tdd and unit testing: Piling a trend upon a trend. *ACM Inroads*, 3(3):62–68, Sept. 2012.
[50] D. Janzen and H. Saiedian. Test-driven learning in early programming courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*, pages 532–536, New York, NY, USA, 2008. ACM.
[51] D. S. Janzen, J. Clements, and M. Hilton. An evaluation of interactive test-driven labs with webide in cs0. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*, pages 1090–1098, Piscataway, NJ, USA, 2013. IEEE Press.
[52] D. S. Janzen and H. Saiedian. Test-driven learning: Intrinsic integration of testing into the cs/se curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*, pages 254–258, New York, NY, USA, 2006. ACM.
[53] D. S. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 719–722, Washington, DC, USA, 2007. IEEE Computer Society.
[54] C. Johnson. Speccheck: Automated generation of tests for interface conformance. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*, pages 186–191, New York, NY, USA, 2012. ACM.
[55] E. L. Jones. Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00)*, pages 153–157, New York, NY, USA, 2000. ACM.
[56] E. L. Jones. Integrating testing into the curriculum – arsenic in small doses. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*, pages 337–341, New York, NY, USA, 2001. ACM.
[57] V. Lustosa Neto, R. Coelho, L. Leite, D. Guerrero, and A. Mendonca. Popt: A problem-oriented programming and testing approach for novice students. In *International Conference on Software Engineering (ICSE)*, pages 1099–1108, May 2013.
[58] C. MacNish. Evolutionary programming techniques for testing students' code. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00)*, pages 170–173, New York, NY, USA, 2000. ACM.
[59] D. J. Malan. Cs50 sandbox: Secure execution of untrusted code. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, pages 141–146, New York, NY, USA, 2013. ACM.
[60] S. A. Markham and K. N. King. Using personal robots in cs1: Experiences, outcomes, and attitudinal influences. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '10)*, pages 204–208, New York, NY, USA, 2010. ACM.
[61] W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)*, pages 4–8, New York, NY, USA, 2005. ACM.
[62] M. Morisio, M. Torchiano, and G. Argentieri. Assessing quantitatively a programming course. In *International Symposium on Software Metrics*, pages 326–336, Sept 2004.
[63] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08)*, pages 68–77, Swinton, UK, UK, 2008. British Computer Society.
[64] M. Pozenel, L. Furst, and V. Mahnicc. Introduction of the automated assessment of homework assignments in a university-level programming course. In *38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 761–766, May 2015.
[65] V. K. Proulx. Test-driven design for introductory oo programming. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)*, pages 138–142, New York, NY, USA, 2009. ACM.
[66] A. Radermacher and G. Walia. Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*, pages 525–530, New York, NY, USA, 2013. ACM.
[67] S. Rahman. Applying the tbc method in introductory programming courses. In *Annual Frontiers In Education Conference (FIE)*, pages T1E–20–T1E–21, Oct 2007.

[68] M. Ricken and R. Cartwright. Test-first java concurrency for the classroom. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*, pages 219–223, New York, NY, USA, 2010. ACM.

[69] M. Rubio-SÃ¡nchez, P. Kinnunen, C. Pareja-Flores, and . VelÃ¡zquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 31(1):453–460, 2014.

[70] C. Schulte and J. Bennedsen. What do teachers teach in introductory programming? In *Proceedings of the Second International Workshop on Computing Education Research (ICER '06)*, pages 17–28, New York, NY, USA, 2006. ACM.

[71] Z. Shams. Automated assessment of students' testing skills for improving correctness of their code. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH '13)*, pages 37–40, New York, NY, USA, 2013. ACM.

[72] Z. Shams and S. H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*, pages 53–58, New York, NY, USA, 2013. ACM.

[73] Z. Shams and S. H. Edwards. Checked coverage and object branch coverage: New alternatives for assessing student-written tests. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE'15)*, pages 534–539, New York, NY, USA, 2015. ACM.

[74] J. Snyder, S. H. Edwards, and M. A. Pérez-Quiñones. Lift: Taking gui unit testing to new heights. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*, pages 643–648, New York, NY, USA, 2011. ACM.

[75] D. Souza, J. Maldonado, and E. Barbosa. Progtest: An environment for the submission and evaluation of programming assignments based on testing activities. In *IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10, May 2011.

[76] J. Spacco and W. Pugh. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pages 907–913, New York, NY, USA, 2006. ACM.

[77] J. Spacco, W. Pugh, N. Ayewah, and D. Hovemeyer. The marmoset project: An automated snapshot, submission, and testing system. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pages 669–670, New York, NY, USA, 2006. ACM.

[78] M. Thornton, S. Edwards, and R. Tan. Helping students test programs that have graphical user interfaces. In *International Multi-Conference on Society, Cybernetics and Informatics (IMSCI)*, pages 164–169, 2007.

[79] M. Thornton, S. H. Edwards, R. P. Tan, and M. A. Pérez-Quiñones. Supporting student-written tests of gui programs. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*, pages 537–541, New York, NY, USA, 2008. ACM.

[80] V. Thurner and A. Bottcher. An "objects first, tests second" approach for software engineering education. In *IEEE Frontiers in Education Conference (FIE)*, pages 1–5, Oct 2015.

[81] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4fun: A web-based environment for educational gaming via automated test generation. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 730–733, Nov 2013.

[82] S. A. Turner. Looking glass: A c++ library for testing student programs through reflection. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*, pages 528–533, New York, NY, USA, 2015. ACM.

[83] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*, pages 19–26, New York, NY, USA, 2014. ACM.

[84] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*, pages 117–122, New York, NY, USA, 2013. ACM.

[85] J. L. Whalley and A. Philpott. A unit testing approach to building novice programmers' skills and confidence. In *Proceedings of the Thirteenth Australasian Computing Education Conference (ACE '11)*, pages 113–118, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[86] M. Wick, D. Stevenson, and P. Wagner. Using testing and junit across the curriculum. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*, pages 236–240, New York, NY, USA, 2005. ACM.

[87] H. Zhang, M. A. Babar, and P. Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, June 2011.