Software Testing in Introductory Programming Courses: A Systematic Mapping Study

Conference Paper · February 2019

DOI: 10.1145/3287324.3287384

4 authors, including:

Lilian Passos Scatalon
University of São Paulo
18 PUBLICATIONS **95** CITATIONS
SEE PROFILE

Rogério Eduardo Garcia
São Paulo State University
85 PUBLICATIONS **461** CITATIONS
SEE PROFILE

Ellen Barbosa
University of São Paulo
201 PUBLICATIONS **1,574** CITATIONS
SEE PROFILE

Software Testing in Introductory Programming Courses

A Systematic Mapping Study

Lilian Passos Scatalon
University of São Paulo (ICMC-USP)
São Carlos-SP, Brazil
lilian.scatalon@usp.br

Jeffrey C. Carver
University of Alabama
Tuscaloosa-AL, United States
carver@cs.ua.edu

Rogério Eduardo Garcia
São Paulo State University (FCT-Unesp)
Presidente Prudente-SP, Brazil
rogerio.garcia@unesp.br

Ellen Francine Barbosa
University of São Paulo (ICMC-USP)
São Carlos-SP, Brazil
francine@icmc.usp.br

ABSTRACT

Traditionally, students learn about software testing during intermediate or advanced computing courses. However, it is widely advocated that testing should be addressed beginning in introductory programming courses. In this context, testing practices can help students think more critically while working on programming assignments. At the same time, students can develop testing skills throughout the computing curriculum. Considering this scenario, we conducted a systematic mapping of the literature about software testing in introductory programming courses, resulting in 293 selected papers. We mapped the papers to categories with respect to their investigated topic (curriculum, teaching methods, programming assignments, programming process, tools, program/test quality, concept understanding, and students' perceptions and behaviors) and evaluation method (literature review, exploratory study, descriptive/persuasive study, survey, qualitative study, experimental and experience report). We also identified the benefits and drawbacks of this teaching approach, as pointed out in the selected papers. The goal is to provide an overview of research performed in the area, highlighting gaps that should be further investigated.

CCS CONCEPTS

• **Social and professional topics → Computing education**; *CS1*;
• **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Software testing; introductory programming courses; systematic mapping

ACM Reference format:
Lilian Passos Scatalon, Jeffrey C. Carver, Rogério Eduardo Garcia, and Ellen Francine Barbosa. 2019. Software Testing in Introductory Programming Courses. In *Proceedings of Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, February 27-March 2, 2019 (SIGCSE '19),* 7 pages.
https://doi.org/10.1145/3287324.3287384

1 INTRODUCTION

Because industrial software practitioners interact with testing, whether or not they hold a 'testing' job, they need to acquire testing skills [6]. Even so, we are graduating students from computing programs who have deficiencies in software testing skills [3, 12].

A way to deal with this issue is to address software testing earlier in the computing curriculum, beginning in introductory programming courses [4]. The idea is to provide students the opportunity to develop their testing skills incrementally throughout the curriculum [9]. Moreover, knowledge of testing can help students improve their programming skills [5, 8, 14].

However, the integration of software testing in this context is not straightforward, since there are many different ways to design the introductory programming sequence [10, 13]. The ACM/IEEE CS curricular guidelines point out some design aspects that vary in these courses, such as the institutional context (with different target audiences), the choice of programming paradigm and language, the platform (e.g. desktop/laptop, web, mobile devices), and the use of software development practices to support the programming activities (e.g. version control, refactoring, and design patterns) [1].

In this paper, our goal is to investigate the integration of testing into this diverse context and provide an overview of the research performed in the area. We conducted a systematic mapping of the literature [11], which resulted in 293 selected papers. We mapped the studies to categories of investigated topic and evaluation method.

Our results highlight elements that compose an integrated *teaching method* of programming and testing (course materials, programming assignments, programming process, supporting tools) and the *learning outcomes* that have been investigated in the area (program/test quality, concept understanding and students' perceptions and behaviors). We also discuss the benefits and drawbacks identified in the selected papers, aiming to help with trade-off decisions in curriculum/course design and with the conduction of further research in the area.

The remainder of this paper is organized as follows. Section 2 describes our research questions and the protocol we followed to conduct the systematic mapping. Section 3 presents the selected studies and provides answers to the proposed research questions. Section 4 discusses the obtained results and, finally, Section 5 presents conclusions, threats to validity and provides directions to future work.

2 RESEARCH METHOD

We followed the guidelines of Petersen et al. [11] to define the research protocol and to conduct the study. Briefly, we performed the following steps:

- definition of review scope (Section 2.1);
- search and selection of relevant papers (sections 2.2 and 2.3);
- definition of a classification scheme, composed by the categories for the mapping (Section 2.4), and
- data extraction from selected papers and mapping to the defined categories (Section 2.5).

2.1 Research Questions

To scope the study, we defined the following research questions:

RQ1: Which topics have researchers investigated about software testing in introductory programming courses?

RQ2: What are the benefits and drawbacks about the integration of software testing into introductory programming courses?

2.2 Search Strategy

We conducted the search for relevant papers in two steps: (i) an automatic search in databases, which provided a list of relevant papers and (ii) a backward snowballing from this preliminary list to identify additional relevant papers [15].

We performed the automatic search in five databases: ACM Digital Library[1], IEEEXplore[2], ScienceDirect[3], Scopus[4], Springer Link[5]. We selected these databases because they are among the most used ones by previous systematic reviews [16].

We constructed the search string following the approach by Zhang et al. [16]. We piloted a previous version of this protocol and formed a reference list composed of 158 papers. Since there was a high variability in the expressions authors use to refer to the teaching of programming and software testing in this context, we performed a frequency analysis of individual words from the titles, abstracts, and keywords of our reference list. We chose the most frequent ones that were able to retrieve all papers from the reference list and arranged them along three aspects: programming, testing and educational context.

The results of that process produced the following search string that we executed in the search engines:

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

[1] http://dl.acm.org
[2] http://ieeexplore.ieee.org
[3] http://sciencedirect.com
[4] http://scopus.com
[5] http://link.springer.com

2.3 Selection criteria

The included papers discuss or investigate software testing in the context of teaching programming fundamentals in higher education, according to the scope defined by the research questions.

Once we had all papers returned from the automatic search, we excluded duplicate papers and papers not written in English. Also, we excluded papers whose context was outside higher education or that only addressed advanced computing courses.

Finally, following a similar approach to Radermacher and Walia [12], we only selected papers since 2000, because papers published earlier than that would not represent current educational practices.

2.4 Classification Scheme

The classification scheme refers to the categories to which we mapped the selected papers. We defined categories for two facets: *investigated topic* and *evaluation method.*

The structure of **investigated topics** provides an overview of the area, helping to answer RQ2. We defined the categories by following the approach of keywording, as suggested by Petersen et al. [11]. Briefly, we looked for concepts that represented the contribution of each paper. Then, we combined these identified concepts in order to form the categories. The idea was to identify categories which would accommodate all selected studies.

For **evaluation method**, we adopted the categories used by Al-Zubidy et al. [2] in their review of Computer Science Education studies, since our mapping is within the scope of theirs. Namely, **literature review** (a review of existing studies in a given topic), **exploratory study** (involves observation and model building), **descriptive/persuasive study** (an overview of the current situation in a given topic), **survey** (subjects are surveyed about some intervention), **qualitative study** (involves the analysis of qualitative data), **experimental** (includes experiments, quasi-experiments and case studies), **experience report** (not a planned study, a report about the experience of applying an intervention) and **not applicable** (a proposal, but without an evaluation).

2.5 Data extraction

We extracted the following elements from each selected paper: year; publication venue (journal/conference); evaluation method; investigated topic; and benefits and drawbacks of software testing in introductory programming courses. One of the authors (Scatalon) did the reading and extraction of these elements for the selected papers.

3 RESULTS

Figure 1 shows the results of the search for relevant papers. The automatic search returned 9091 studies in total, from which we selected 229 relevant studies by applying the selection criteria. Next, we applied backward snowballing and obtained 64 additional relevant studies, arriving at a total of 293 selected papers.

As Table 1 shows, the selected papers appear in a wide variety of conferences and journals. We listed the more frequent ones. Most studies were published in venues about CS Education (SIGCSE,

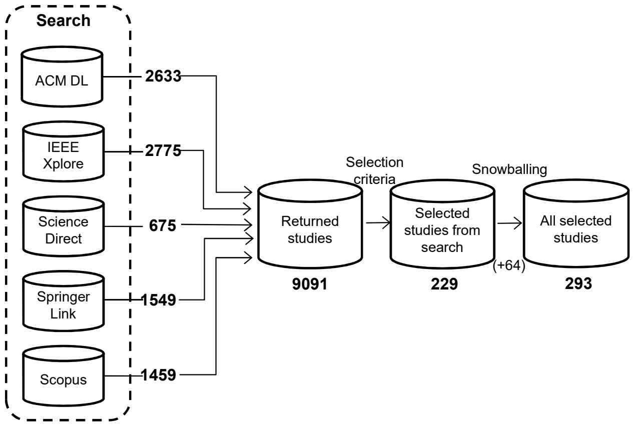

Figure 1: Results

ITiCSE, ACE, ICER, Koli Calling, SIGCSE Bulletin, Computer Science Education), followed by venues that address Software Engineering Education (ICSE and CSEE&T), education in computing-related curricula (FIE, Journal of Computing Sciences in Colleges, ACM TOCE), and, finally, venues with a more general focus in technology in education (L@S and Computers & Education).

The following subsections provide an analysis of the identified papers. Due to the large number of papers included in this mapping study, we are not able to include all references in the paper. A full list of papers along with the corresponding study number can be found at http://bit.ly/SIGCSE19-SM-Testing. This list of references is grouped based on the topics described in the next section.

Table 1: Distribution of publication venues

Venue Name	Venue Type	#
SIGCSE	conference	49
ITiCSE	conference	37
Journal of Computing Sciences in Colleges	journal	34
FIE	conference	23
OOPSLA/SPLASH	conference	13
ICSE	conference	10
CSEE&T	conference	9
ACE	conference	9
SIGCSE Bulletin	journal	9
ICER	conference	6
Koli Calling	conference	5
L@S	conference	4
Computer Science Education	journal	3
Software: Practice and Experience	journal	3
ACM JERIC/TOCE	journal	2
Computers & Education	journal	2
other		75
total		293

3.1 RQ1: Investigated topics

We identified nine *investigated topics* in the selected papers. Figure 2 shows the map of selected papers to the categories of topic and evaluation method. The topic **curriculum** includes papers about the integration of testing in the computing curriculum as a whole or in individual programming courses.

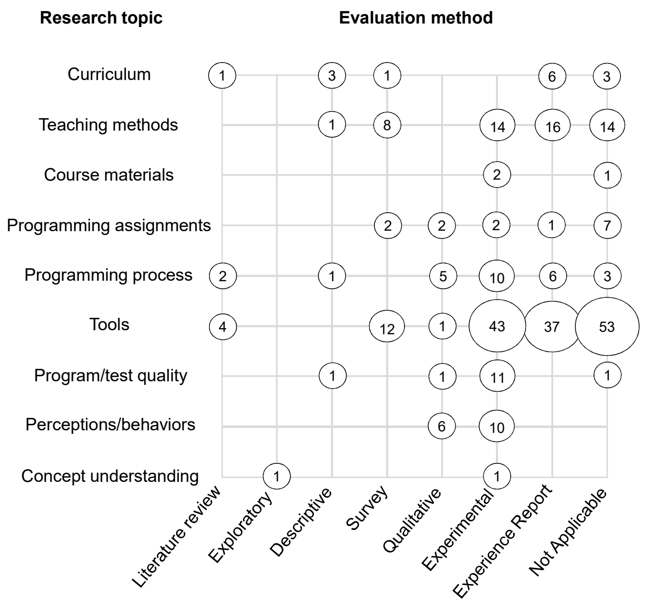

Figure 2: Map of research on software testing in introductory programming courses

Teaching methods include methods to teach programming with the integration of software testing. We identified general elements that compose a teaching method in this scenario. We also considered these elements as topics in our study: **course materials** (materials about testing for the context of introductory courses), **programming assignments** (guidelines to conduct programming assignments that include testing practices), **programming process** (programming processes for novices), and **tools** (supporting tools). When a selected paper addressed more than one of these elements, we mapped it to the topic *teaching methods*. Otherwise, it was mapped to the topic of the corresponding element.

The remaining topics concern the learning outcomes of the integration of testing in programming courses: **program/test quality** (assessment of students' submitted code), **perceptions/ behaviors** (students' attitudes towards software testing) and **concept understanding** (assessment of students' knowledge of programming and testing concepts). Next we provide an overview of the selected papers according to the investigated topics.

3.1.1 Curriculum. The papers mapped to the topic *curriculum* recommend testing concepts and practices should be distributed throughout the computing curriculum (S3, S5, S11, S12, S13). Moreover, the idea is to address testing earlier, beginning in introductory programming courses, by integrating testing practices into programming assignments (S1). Some papers address the design of a specific programming course with the integration of testing, such as S2, S4, S6 and S8.

3.1.2 Teaching methods. The papers mapped to this topic propose or investigate methods to teach programming with the integration of software testing, i.e. the different ways to teach these two subjects together in this context. Janzen and Saiedian (S35, S36) proposed *Test-Driven Learning* (TDL), which is a method to teach programming by introducing new concepts through unit tests. The authors provide several guidelines on how to apply it in the classroom. Edwards (S33, S17, S34) proposed the combination of TDD and the use of an automated assessment tool (Web-CAT) to leverage constant feedback as students submit their programs and test suites.

3.1.3 Course materials. There are only three selected papers investigating course materials about software testing that can be used in the context of introductory courses. Agarwal et al. (S68) and Barbosa et al. (S70) presented educational modules of software testing. They linked the materials according to the instructional sequence in which novice programmers should learn testing concepts. Desai et al. (S69) demonstrated how to adjust existing materials from a programming course to integrate software testing. They also report about their experience to apply the materials in the classroom.

3.1.4 Programming assignments. Papers about this topic discuss guidelines to design, conduct, and assess programming assignments that include testing practices. Some selected papers present descriptions of particular assignments (e.g. nifty assignments or programming projects) and include information about the appropriate context to apply them (S75, S78, S79, S80).

The design of assignments involving testing includes some additional important aspects to consider. First, there is the need to decide whether students should write test cases or work with instructor's tests (S77). Second, the problem specification should be clear enough so students are able to write tests (S74, S76).

Finally, testing is an inherent part of assessing students' programs, by providing a metric of correctness (S84). It can ease the grading process, especially when using an automated assessment tool. Tools can provide adequate results and feedback for formative assessment (such as homeworks and lab sessions) but might be less feasible for summative assessment (tests and exams), according to S72.

3.1.5 Programming process. Several proposals aim to teach students a systematic approach to develop programs, which can be seen as a lightweight version of a software development process. Given the scope of the systematic mapping, all processes addressed in the selected papers involve software testing, binding it somehow with programming.

The programming process can be easily overlooked in introductory courses, especially when students learn programming mostly by seeing examples of ready-made solutions. Instead, they should also learn how to stepwise create a solution for a given problem and to reflect about their own development process (S110).

Several studies investigate the use of **TDD** (*test-driven development*) by novice programmers (e.g. S92, S94, S96, S98, S99, S102, S103). There are other proposals of programming process, specifically designed for the educational context, which are also heavily influenced by TDD. Some examples are TBC (*Testing Before Coding*), POPT (*Problem-Oriented Programming and Testing*) and STREAM (Stubs, Tests, Representations, Evaluation, Attributes and Methods) from S87, S97 and S89, respectively.

Usually these processes address the testing activity from a high-level point of view, without giving details about how students should test their programs. In another perspective, two studies investigated the testing activity for novice programmers at a lower-level, focusing on test design and the use of testing criteria (S98 and S101).

3.1.6 Tools. The papers mapped to this topic present several kinds of tools, which automate different aspects of applying testing practices. We sort them into three groups:

- **Supporting mechanisms to write and execute test cases:**
  - **Testing frameworks/libraries:** besides the xUnit testing frameworks, there are testing libraries developed specifically to ease the learning curve for students (e.g. S201, S153, S226, S178).
  - **IDEs' testing facilities:** there are IDEs that offer mechanisms to help students test their programs, like BlueJ (S204).
- **Automated assessment systems:** automating the assessment process by means of software testing is fairly straightforward, since test cases are represented by code that can be executed along with submitted programs. In this sense, *automated assessment* permeates most tools used in this context. We sort them into the following categories:
  - **Submission and testing systems:** These systems usually are responsible for compiling the submitted program, executing tests and providing feedback to students. In some cases these tools also grades students' submitted code according to test results in a (semi-)automatic manner. In general, these systems are web-based (S254, S215, S138, S177) or plug-ins to other widely used systems such as IDEs (S208) or LMSs (S145).
  - **Online judges:** These tools present a catalog of problems to students, who should submit the corresponding programming solutions to be assessed by means of testing. Some of these systems are used in programming competitions (S133, S176).
  - **Games:** Some tools can be characterized as games, which aim to motivate students through fun and competition. They introduce software testing in different ways, such as implicit testing to solve programming "quests" (S223), or hints in the format of unit tests, which help students to guess a "secret implementation" in code duels (S142).
  - **Tutor systems:** Some tools combine course materials and interactive exercises or assignments, providing automatic orientation while students learn programming and testing. Usually this kind of tools is composed by materials (presented as slides or hypertext) and an automated assessment tool to test students' programs (S118, S121, S123, S167).

- **Automated assessment utilities**: Some papers focus on functionalities that compose or complement automated assessment systems. We provide an overview of these proposals, sorting them into the different aspects they address:
  - **Test automation:** There are several proposals that aim to make the execution of tests and students' programs as seamless as possible. One important aspect in this scenario is *interface conformance* between program and test suite (S258, S164, S241), seeking to assure that both are compiled together properly. Additionally, some precautions should be taken during execution, like running students' code in a sandbox to assure safety and having mechanisms to cope with infinite loops (S194, S174).
  - **Feedback:** Metrics of program and test quality can be used to suggest a grade and to provide feedback to students. Besides that, students need help with failed test cases and improving their test suites. Some tools provide this type of additional support to students, like mechanisms to detect inadequate memory management (S154) and the generation of execution traces (S131). Feedback can also help to influence students' testing behavior, with the adaptive release of hints as students achieve certain testing goals (S173, S179).

3.1.7 Program/test quality. Papers mapped to this topic address students' performance in programming assignments, assessed by means of their submitted code (program and/or tests). The program usually is assessed in terms of **correctness**, which in turn is calculated by the **success rate** of a given test suite. Besides correctness, there are also metrics that involve a static analysis of the source code structure, by analyzing modifications made by students between successive submissions (S270, S273).

When students are supposed to write tests in the assignments, the issue of assessing the quality of their test suites is raised. The most common metric is **code coverage**, which generates feedback that is easy for students to understand. However, code coverage can overestimate test quality, since it is possible to achieve 100% coverage even when a test suite is not thorough, e.g. when the tests do not check for missing features in the program. In this sense, other strategies like **mutation analysis** and **all-pairs testing** can provide more accurate metrics (S264, S266), though both are more computationally expensive.

3.1.8 Concept understanding. There are only two selected studies that address this topic (S277, S278). Both aim to investigate assessments of programming concepts, which include software testing concepts. Sanders et al. (S277) presented the Canterbury QuestionBank[6], a repository of 654 multiple choice questions about programming fundamentals, 3% of which about testing. Luxton-Reilly et al. (S278) present a comprehensive review of concepts that should be assessed in introductory programming courses. *Testing* appears under the category of programming process, along with other topics like debugging and design.

3.1.9 Students' perceptions and behaviors. Papers mapped to this topic investigate students' attitudes towards software testing. Students' perceptions indicate their opinions about the testing approaches, such as TDD acceptance (S280). Students' behaviors refer to what they actually do during programming assignments, in contrast with what they were instructed to do (S289, S291, S292, S293).

We can observe trends in student behavior when analyzing submissions of the whole class (S284, S282), such as "happy-path" testing in S286. There are also studies analyzing multiple subsequent submissions for a given assignment, i.e. "snapshots" of students' programs and test suites. Process adherence (e.g. whether students are adopting test-first or not) and mechanisms to influence students' behavior can be investigated using this strategy (S280, S287, S281).

3.2 RQ2: Benefits and Drawbacks

To answer RQ2, we identified benefits and drawbacks of integrating software testing into programming courses, as pointed out by the selected papers. We identified the following **benefits**:

- **Improvement in students' programming performance**: there are studies reporting improvements in students' performance, mainly in terms of program quality (S33, S101, S276, S272). There are also findings indicating improvements in the resulting program design (in S94, for TDD specifically). The papers argue the reason behind these improvements is that testing practices help developing students' comprehension and analysis skills (S34).
- **Feedback**: test results can provide students useful information about their programming and testing performance before the assignment deadline (S21, S34, S92, S119, S210). This issue of providing feedback to students is recurrent in the motivation for using automated assessment tools (S112 to S261). Automated feedback may decrease the amount of help students need from the instructor (S150). Moreover, since testing drives students to self-validate their work, it can help them make progress when they are stuck or recognize when they need help from the instructor (S62).
- **Objective assessment**: testing results provide an objective and consistent way to assign grades to the assignments (S192). This benefit of testing is also frequently discussed in the selected papers about supporting tools (S112 to S261). Besides helping in the grading process, the assessment through testing also helps students to better understand the correctness requirements of assignments (S62).
- **Better understanding of the programming process**: when software testing is introduced, students learn a simplified version of the development process (S89). Considering that they have to work on many programming assignments throughout the introductory sequence, students have an opportunity to learn the mechanics of the activities of programming and testing together (S104).

Conversely, we identified the following **drawbacks**:

- **Additional workload of course staff**: instructors and teaching assistants may have additional work to adjust course materials to include testing concepts, prepare reference test suites for assignments, and assess students' test suites (S33, S17, S62, S72).
- **Students' testing performance**: studies report the lack of proper testing by students. Students mostly check common

[6] web-cat.org/questionbank

program behavior, leaving out corner cases that would be crucial to reveal the presence of defects (S286). In order to be properly understood and applied, many testing ideas require previous knowledge and skill in programming, which students are also still acquiring in programming courses. In particular, the main difficulty may be related to students writing their own test cases (S38, S62).
- **Students' reluctance to conduct testing**: students may present a negative attitude towards software testing, even though they recognize the importance of the testing activity (S61). When the testing practice is voluntary, they may not develop test cases for their programs (S32).
- **Programming courses are already packed**: the integration of software testing brings the need to cover additional topics in courses that may be already full. In other words, it is the integration of additional content with the same amount of lecture hours (S123, S17, S34).

4 DISCUSSION

The map of papers in Figure 2 allows us to analyze the distribution of research in the area. In terms of investigated topics, over half of papers are about *tools*, 51.19% (150). Conversely, the topics *course materials* (1.02% – 3) and *concept understanding* (0.68% – 2) cover only a small amount of research performed in the area.

These less investigated topics can indicate areas in need of more research, which could be directed to minimize the identified drawbacks in Section 3.2. For example, if more course materials were available, it could help to minimize the additional workload of course staff. Also, these materials could help identify ways to integrate testing without disrupting programming courses. Similarly, more research in teaching novice's *programming process* and in fostering *concept understanding* (especially basic testing concepts) could help to improve students' testing performance and their reluctance to perform software testing.

As to evaluation methods, selected papers that present empirical studies (*survey*, *qualitative* and *experimental*) comprise 44.7% (131) of the studies. Conversely, papers classified as *not applicable* or *experience report* comprise 50.51% (148), slightly above half of selected papers. The problem with the latter kind of studies is the lack of empirical evidence. Papers in *not applicable* present only proposals with no evaluation. *Experience reports* are not planned studies, so the conclusions rely on the researcher's perceptions.

Still considering the distribution of research in the area, the investigation of TDD is noteworthy throughout the selected papers, because 27% (89) mention it. For the topic *curriculum*, Edwards (S1) argues that TDD should be used in all programming assignments of the computing curriculum, from the CS1 course. Adams (S2) advocates that it should start in the CS2 course instead. In *teaching methods*, Edwards (S33, S17, S34, S21) proposed and investigated the use of TDD combined with an automated assessment tool in the classroom. In *course materials*, Desai et al. (S69) showed how TDD can be integrated into existing course materials, considering the same amount of lecture hours and without reducing the coverage of programming topics. Marrero and Settle (S71) discussed *assignments* with TDD in the context of two different programming courses. Spacco and Pugh (S279), Janzen and Saiedian (S280), and Buffardi and Edwards (S281, S285) studied mechanisms to motivate students to apply TDD. Besides these papers that investigate TDD specifically, it is possible to see the TDD's influence on the definition of other proposals (e.g. S87, S97, S89).

5 CONCLUSIONS

In this paper we provide an overview of the research performed about the integration of software testing into introductory programming courses. We conducted a systematic mapping study, which resulted in 293 selected papers. We classified papers according to investigated topic and evaluation method. We also discussed benefits and drawbacks of the approach.

There is a wide variety of ways to integrate testing into introductory courses, as can be observed in the selected papers. We identified a structure of topics (Section 3.1) which outlines a *teaching method* of both subjects (course materials, programming assignments, programming process and tools) and the corresponding students' learning outcomes (program/test quality, perceptions/behaviors and concept understanding). These identified elements can be seen as variables involved in the integration of testing in programming courses.

Similarly, the identified benefits and drawbacks in Section 3.2 motivate further research, aiming to raise the benefits and minimize the drawbacks. Also, as discussed in Section 4, the distribution of research over the identified topics shows a high concentration of papers about supporting tools (above half of the selected papers). This may indicate researchers the need to consider other topics in the area as well.

In terms of evaluation method, our results show a high number of studies (slightly above half of the papers) that are either experience reports or proposals with no evaluation (classified as "not applicable"). This result means a significant amount of studies lack a planned evaluation. This practice is concerning, since there is the need to transition from this kind of study to generating theory through empirical studies, by generating and iterating on hypotheses in CS Education research [7].

This systematic mapping contains the following threats to validity. First, it is important to consider that the first author individually performed all the steps of this study, including the selection of papers, reading, definition of classification scheme, extracting data, and mapping of the studies. For validation, the other authors provided feedback on all steps. Second, piloting the protocol helped to minimize biases in this study results. Third, we limited the publication date of selected papers to after the year 2000. Again, we believe that papers before that would not represent current educational approaches, specially in a area such Computer Science.

As future work, we intend to investigate in detail the variables involved in the integration of testing into programming courses. More specifically, we intend to conduct a systematic review of a subgroup of the selected papers: the empirical studies (papers mapped to *survey*, *qualitative* and *experimental*). Our goal is to help in the designing and planning of future studies, which in turn can help to further investigate topics lacking of research.

ACKNOWLEDGMENTS

This work was supported by FAPESP (State of São Paulo Research Funding Agency) grants 2014/06656-8 and 2016/17575-4.

REFERENCES

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

[2] Ahmed Al-Zubidy, Jeffrey C. Carver, Sarah Heckman, and Mark Sherriff. 2016. A (Updated) Review of Empiricism at the SIGCSE Technical Symposium. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, NY, USA, 120–125. https://doi.org/10.1145/2839509.2844601

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

[4] Stephen H. Edwards. 2003. 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)*. ACM, New York, NY, USA, 148–155. https://doi.org/10.1145/949344.949390

[5] Stephen H. Edwards. 2004. 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)*. ACM, New York, NY, USA, 26–30. https://doi.org/10.1145/971300.971312

[6] Vahid Garousi and Junji Zhi. 2013. A Survey of Software Testing Practices in Canada. *Journal of Systems and Software* 86, 5 (May 2013), 1354–1376.

[7] Mark Guzdial. 2013. Exploring Hypotheses About Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 19–26.

[8] David Janzen and Hossein Saiedian. 2008. Test-driven Learning in Early Programming Courses. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 532–536. https://doi.org/10.1145/1352135.1352315

[9] Edward L. Jones. 2001. Integrating testing into the curriculum – arsenic in small doses. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. ACM, New York, NY, USA, 337–341.

[10] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A Survey of Literature on the Teaching of Introductory Programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '07)*. ACM, New York, NY, USA, 204–223. https://doi.org/10.1145/1345443.1345441

[11] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08)*. British Computer Society, Swinton, UK, UK, 68–77.

[12] Alex Radermacher and Gursimran Walia. 2013. Gaps Between Industry Expectations and the Abilities of Graduates. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 525–530.

[13] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. 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)*. ACM, New York, NY, USA, 19–26.

[14] Jacqueline L. Whalley and Anne Philpott. 2011. A Unit Testing Approach to Building Novice Programmers' Skills and Confidence. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE '11)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 113–118. http://dl.acm.org/citation.cfm?id=2459936.2459950

[15] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. ACM, New York, NY, USA, Article 38, 10 pages.

[16] He Zhang, Muhammad Ali Babar, and Paolo Tell. 2011. Identifying relevant studies in software engineering. *Information and Software Technology* 53, 6 (2011), 625 – 637. Special Section: Best papers from the APSEC.