UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

A framework for experimental studies on the integration of software testing into programming education

Lilian Passos Scatalon

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: ______________________

Lilian Passos Scatalon

A framework for experimental studies on the integration of software testing into programming education

Thesis submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *EXAMINATION BOARD PRESENTATION COPY*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Dr. Ellen Francine Barbosa

USP – São Carlos
February 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

Scatalon, Lilian Passos
S287f A framework for experimental studies on the
integration of software testing into programming
education / Lilian Passos Scatalon; orientadora
Ellen Francine Barbosa. -- São Carlos, 2019.
200 p.

Tese (Doutorado - Programa de Pós-Graduação em
Matemática) -- Instituto de Ciências Matemáticas e
de Computação, Universidade de São Paulo, 2019.

1. Programming education. 2. Software testing.
3. Experimental studies. 4. Experimental framework.
I. Barbosa, Ellen Francine, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Lilian Passos Scatalon

Um framework para estudos experimentais sobre a integração de teste de software no ensino de programação

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutora em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Dr. Ellen Francine Barbosa

USP – São Carlos
Fevereiro de 2019

“After reflecting on the first experiments conducted, one is often jarred at the end of an investigation by (and even a little ashamed of) how pathetic they were. The wrong variables may have originally been examined or important variables may have been investigated, though far outside the right region. It is like watching a film about a swimmer, who now somersaults from the springboard, when he was only a small boy learning how to swim. It would be ridiculous to start by doing somersaults and neurotic to say "if I cannot somersault from the springboard now, I prefer not to learn how to swim". Researchers must learn from the swimmer, who was ready to put his foot in the water and not afraid of getting wet.”

Natalia Juristo and Ana M. Moreno
Basics of Software Engineering Experimentation
page 48

RESUMO

SCATALON, L. P. **Um framework para estudos experimentais sobre a integração de teste de software no ensino de programação**. 2019. 200 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Disciplinas introdutórias de programação compõem o núcleo de diversos cursos de graduação, visto que se trata de uma habilidade crucial para profissionais em muitas áreas de ciências exatas. Buscando lidar com as dificuldades de aprendizagem dos alunos nessas disciplinas, os professores podem adotar diferentes abordagens de ensino, uma vez que há muitas variantes no ensino de programação (como linguagens e paradigmas de programação, práticas de desenvolvimento, plataformas, ferramentas de apoio, etc). Em particular, a abordagem de ensino que consiste em integrar teste de software nesse contexto tem se destacado na área, pois pode levar os alunos a pensarem de modo mais crítico enquanto resolvem atividades práticas de programação. Mesmo assim, essa abordagem de ensino também pode apresentar desafios significativos, como a resistência dos alunos para conduzir práticas de teste. Nesse sentido, estudos experimentais têm o papel de fornecer evidência acerca de resultados em termos de aprendizagem, considerando diferentes abordagens de ensino e contextos. Porém, estudos na área de ensino de programação muitas vezes apresentam falta de fundamentação teórica, ou seja, não são construídos a partir de teorias, modelos e frameworks estabelecidos na área. Isso significa que os aspectos (ou variáveis) utilizados para investigar as abordagens de ensino não são adequadamente caracterizados nos estudos, o que leva a dificuldades em interpretar os resultados obtidos e construir conhecimento na área. Como consequência, os professores são impedidos de ter evidências confiáveis para fazer escolhas informadas nas abordagens de ensino utilizadas em sala de aula. Considerando esse cenário, este trabalho de doutorado propõe o uso de modelos de domínio para apoiar pesquisadores ao definir e projetar experimentos no ensino de programação. Mais especificamente, o domínio da abordagem de integração de teste de software foi explorado neste trabalho, com a criação de um *framework* para estudos experimentais sobre a integração de teste de software no ensino de programação. O *framework* provê uma estrutura básica de estudos experimentais nesse domínio, sendo composto por modelos de variáveis relacionadas a essa abordagem de ensino. Neste trabalho também foram conduzidos experimentos de acordo com a estrutura do *framework*. A meta deste trabalho é apoiar pesquisadores e professores ao definir e planejar estudos no cenário educacional, em especial os focados em avaliar a integração de teste de software em disciplinas de programação.

Palavras-chave: Ensino de programação, Teste de software, Estudos experimentais, Framework experimental.

ABSTRACT

SCATALON, L. P. **A framework for experimental studies on the integration of software testing into programming education**. 2019. 200 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

Introductory programming courses compose the core of several undergraduate degree programs, since programming is a crucial technical skill for professionals in many STEM areas. Aiming to address students' learning difficulties in these courses, instructors can adopt different teaching approaches, since there are several varying aspects in programming education (e.g. programming languages and paradigms, development practices, platforms, supporting tools etc). In particular, the teaching approach that consists of integrating software testing into this context has been prominent in the area, since it may lead students to think more critically while working on programming assignments. Even so, this teaching approach can also present significant challenges, such as the students' reluctance to conduct testing practices. In this sense, experimental studies have the role to provide evidence about learning outcomes, considering different teaching approaches and contexts. However, studies in the area of programming education often present a lack of theoretical basis, i.e. are not built upon established theories, models and frameworks. In other words, the varying aspects (or variables) used to investigate teaching approaches are not properly characterized in the studies, what leads to difficulties to interpret the obtained results and build knowledge in the area. As a consequence, instructors are prevented from having reliable evidence to make informed choices of teaching approaches used in the classroom. Considering this scenario, we propose in this PhD thesis the use of domain-specific models to support researchers in scoping and designing experiments on programming education. More specifically, we explored the domain of the software testing integration teaching approach, by creating a framework for experimental studies on the integration of software testing into programming education. The framework provides a basic structure of experimental studies in this domain, composed by models of variables related to this teaching approach. We also conducted experiments on the same domain and demonstrated their instantiation into the framework. We intend to support researchers and instructors in the scoping and planning of experimental studies in the educational scenario, specially those aimed at evaluating the integration of software testing into programming courses.

Keywords: Programming education, Software testing, Experimental studies, Experimental framework.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Context

Programming is a crucial technical skill for software-related professionals (LETHBRIDGE, 2000; LETHBRIDGE *et al.*, 2007; SURAKKA, 2007; RADERMACHER; WALIA, 2013). Besides that, programming has been moving towards becoming a basic skill that people should have for their everyday lives, similarly to reading and writing (VEE, 2013; ELOY *et al.*, 2017; GUZDIAL, 2018).

In this sense, there are initiatives to teach programming since the beginning of basic education to children and teenagers, mainly attempting to promote computational thinking skills (GUZDIAL, 2008; RESNICK *et al.*, 2009; BARR; STEPHENSON, 2011; GROVER; PEA, 2013; LYE; KOH, 2014). There are also lifelong learning initiatives to teach programming like the Hour of Code[1], involving millions of people of all ages worldwide (WILSON, 2015). Therefore, there are many different contexts in which the teaching of programming takes place.

Even when focusing on programming courses in higher education, there are still different audiences, such as computing majors and non majors (FORTE; GUZDIAL, 2005). Besides, there are many other sources of variations in the teaching of programming (ACM/IEEE-CS, 2013): the programming paradigm and language adopted to teach programming concepts, the tools used to support the teaching/learning activities, development practices that can be integrated into programming assignments (such as testing and version control) and so on.

Hence, there is the need to check how students respond in each context to the different ways to teach programming. Researchers have been conducting studies with this purpose, which places research on programming education in a prominent position within the scope of Computer Science Education Research (VALENTINE, 2004; SHEARD *et al.*, 2009; LUXTON-REILLY *et al.*, 2018).

---

[1] <https://hourofcode.com>

In particular, there are several studies showing that CS1 students' performance is below expected, both in terms of programming performance and programming concepts understanding (MCCRACKEN *et al.*, 2001; LISTER *et al.*, 2004; MCCARTNEY *et al.*, 2013; UTTING *et al.*, 2013a; TEW; GUZDIAL, 2011). Also, failure and dropout rates in programming courses may reach worrying levels (BEAUBOUEF; MASON, 2005; BENNEDSEN; CASPERSEN, 2007; WATSON; LI, 2014; ZINGARO, 2015; PETERSEN *et al.*, 2016).

In summary, there are many different ways to teach programming and students tend to present learning difficulties in these courses. Therefore, these variations in the teaching of programming should be investigated to find out what configuration works best for each classroom context. In this sense, the role of experimental studies (and empirical studies in general) on programming education is to provide evidence about students' learning outcomes.

Also, empirical studies are the means to generate and test hypotheses about teaching and learning programming, helping to check if the current understanding about the area is correct (FINCHER; PETRE, 2004; GUZDIAL, 2013). The empirical paradigm has been adopted with this purpose by several areas that involve human-based activities, such as Medicine, Software Engineering and Education (BUDGEN *et al.*, 2009).

Among the empirical methods, experiments have the purpose to test hypotheses and help establishing cause-effect relationships between variables of the area. For example, Salleh, Mendes and Grundy (2014) investigated the effect of students' personality traits on their performance while using pair programming. Guzdial (2013) has investigated the influence of media computation on learning effectiveness, gender diversity, retention and plagiarism in introductory courses. Findings of such kind of studies help to understand when and how students learn programming better.

In particular, the integration of software testing is an approach that stands out in programming education (JONES, 2001; BARBOSA *et al.*, 2003; EDWARDS, 2004; JANZEN; SAIEDIAN, 2008; WHALLEY; PHILPOTT, 2011). As with any teaching approach, there are benefits and drawbacks associated with the integration of testing into this context. Therefore, there is the need to investigate how to raise the benefits and minimize the drawbacks in different classroom contexts. The integration of software testing (and any other teaching approach) is not a "silver bullet" to problems in programming education, but informed choices have better chances to improve students' learning outcomes (VIHAVAINEN; AIRAKSINEN; WATSON, 2014).

The findings of empirical studies can help educators with such informed choices. The conducted studies advance the body of knowledge about programming education, by exploring how students' learning take place in real contexts. In other words, it helps curriculum designers and instructors to deliver programming courses as effective as possible in terms of students' learning and motivation.

1.2 Motivation

There are many reviews of existing studies in this area, either exclusively in the teaching of programming (VALENTINE, 2004; SHEARD *et al.*, 2009) or in the wider scope of Computer Science Education Research (RANDOLPH, 2007; MALMI *et al.*, 2010; AL-ZUBIDY *et al.*, 2016), which also includes studies about programming education. The results of all these reviews indicate that, in general, existing studies present a lack of research rigor (FINCHER; PETRE, 2004; PEARS; MALMI, 2009; ROBINS, 2015; LISHINSKI *et al.*, 2016).

Two subproblems related to lack of research rigor also draw attention in the literature: (i) the high percentage of existing empirical studies that are only experience reports (VALENTINE, 2004; RANDOLPH, 2007; PEARS; MALMI, 2009) and (ii) the fact that studies often presents a lack of theoretical basis (BEN-ARI *et al.*, 2004; BERGLUND; DANIELS; PEARS, 2006; SHEARD *et al.*, 2009; MALMI *et al.*, 2010; KOULOURI; LAURIA; MACREDIE, 2014; MALMI *et al.*, 2014).

Valentine (2004) calls experience reports as “Marco Polo” studies (“I went there, and I saw this”). Randolph (2007) describe them as studies providing anecdotal evidence. This kind of empirical study consists in a case report, not planned (“this is what happens in my classroom” (FINCHER; PETRE, 2004)). The problem is that the obtained conclusions are subjective, limited to researcher’s impressions about the intervention applied in the classroom.

Leaning towards more rigorous empiricism, study types like survey, case study and experiment involve careful planning about how data will be collected and analyzed. Generally speaking, planning includes properly scoping and designing the study. In particular, considering experiments, the researcher needs to build and test a model, which is formed by causes and effects of the phenomenon of interest. Such model is designed by the researcher using variables that represent theoretical constructs/concepts in the area.

About study theoretical basis, Malmi *et al.* (2014) specifies that it consists in the application of established theories, models or frameworks to design the study and discuss the obtained results. Some examples are learning theories (construtivism, cognitive load theory, etc), curricular frameworks (e.g. different versions of ACM/IEEE curriculum), Bloom’s taxonomy, domain models, among others.

Hence, the theorerical basis is related to how the study "builds on previous theoretical research or established practice by applying or extending some theory, model or framework" (MALMI *et al.*, 2014). When there is a lack of theoretical basis, "this may lead to a situation where knowledge in the field accumulates only in small isolated areas, which may be referenced as related work but which are not used as a foundation for new research" (MALMI *et al.*, 2014).

Considering again the lack of research rigor in general, a consequence would be the difficulties to "relate the collected data to an underlying theory. The net result is that results are hard to interpret, and studies cannot be compared" (EASTERBROOK *et al.*, 2008). In contrast, "research that is theoretically sound permits generalization of results, it invites comparison between methods and results, and at the same time it makes the limits of the research visible" (SHEARD *et al.*, 2009).

This consequence becomes explicit when studies with the similar goals provide apparently contradictory results, such as the studies of McCracken *et al.* (2001), McCartney *et al.* (2013) and Utting *et al.* (2013a), which are discussed in Section 2.3. The reason for getting such different results, despite the similar goals and context, was the way in which each study was designed.

In this sense, we believe domain-specific mechanisms to help researchers to scope and design experiments can deal with these problems. A good example is the model of variables on pair programming established by Gallis, Arisholm and Dyba (2003). It was developed in the Software Engineering area, but it was also applied in the area of Computer Science Education (SALLEH; MENDES; GRUNDY, 2014).

Indeed, Malmi *et al.* (2014) and Lishinski *et al.* (2016) observe that most models, theories and frameworks used in CSEd Research come from other disciplines. Nelson and Ko (2018) also highlight that the community of researchers in CSEd have limited resources regarding domain-specific theories.

1.3 Objetives

As discussed in the previous sections, the teaching of programming involve several elements, such as staff, programming paradigm and languages, platforms, supporting tools, development practices and so on. These elements can be combined in many ways resulting in several different teaching approaches. Hence, experimental studies in programming education have a very important role, since they check our understanding about how students have been responding to the different teaching approaches in different contexts.

However, existing studies in the area present a lack of research rigor, often consisting of experience reports with anecdotal evidence and/or presenting a lack of theoretical basis. Both problems are related to a poor study design, specially considering the selection of variables to be investigated and the proper control of the remaining envolved variables throughout the conduction of the study.

In this scenario, we aim to investigate and define domain-specific models to support researchers in scoping and designing experiments on the teaching of programming. Hence, the objectives of this PhD work are the following:

- Definition of an explicit framework that represents the underlying structure of experiments from a given domain within the scope of programming education.

- Conduction of experiments in such domain, in order to explore the proposed framework and also refine it.

We chose the domain of the integration of software testing into programming education to achieve our objectives. Ultimately, the goals are to provide supporting resources to researchers that intend to conduct experiments in this domain and also contribute to the area with the design and findings of the conducted experiments.

1.4 Methods

Computer Science Education research has traditionally borrowed methods from other disciplines (ALMSTRUM *et al.*, 2005; MALMI *et al.*, 2010; FINCHER; TENENBERG; ROBINS, 2011; LISHINSKI *et al.*, 2016). In this sense, we chose to borrow methods and guidelines from the Software Engineering area to undertake the objectives of this PhD work. Although the areas of Education and Software Engineering present their own specificities, they also present a high similarity in their experimental methodology, since they involve both technological and social aspects (BUDGEN *et al.*, 2009).

Therefore, in order to define the experimental framework, we used the same strategy of frameworks defined in the Software Engineering area (BASILI; SHULL; LANUBILE, 1999; GALLIS; ARISHOLM; DYBA, 2003; WILLIAMS *et al.*, 2004; MORRISON, 2015): we leveraged the structures used in existing studies of the domain of interest. To this end, we used methods and guidelines, also from SE, to conduct a systematic mapping and review (KITCHENHAM; CHARTERS, 2007; BRERETON *et al.*, 2007; PETERSEN *et al.*, 2008; ZHANG; BABAR; TELL, 2011; WOHLIN, 2014). In order to conduct the proposed experiments, we also applied the experimental process outlined by Wohlin *et al.* (2012) for Software Engineering, using several guidelines provided by Juristo and Moreno (2001) to make decisions concerning each activity of the process.

1.5 Thesis structure

In this chapter we provided a characterization of this PhD thesis. We presented the research context where the PhD work is inserted, the motivation to undertake it in terms of research gaps in the literature, the objectives we intend to achieve with this work and a description of research methods used as a means to do so. The remaining of this PhD thesis is organized as follows:

- In Chapter 2 we provide an overview of programming education, which is the context of this PhD thesis. We discuss the teaching of programming in terms of curricular guidelines, implementation of courses, students’ learning outcomes and teaching approaches. In particular, the integration of software testing, which is one of the presented teaching approaches, is the domain we chose to explore in our proposed experimental framework.

- In Chapter 3 we present concepts from experimentation in Software Engineering that we used as research method to build the proposed research framework and conduct the proposed experiments. To this end, we discuss the process to conduct experiments and existing experimental frameworks in the literature.
- In Chapter 4 we describe a survey that we conducted with graduates from computing undergraduation programs in Brazil, aiming to identify knowledge gaps in software testing caused by how computing curricula have been implemented in the universities. We also discuss how the identified knowledge gaps can be addressed by integrating software testing into programming courses.
- In Chapter 5 we present the results of a systematic mapping that we conducted on the literature about the integration of software testing into programming courses. We identified how instructors have configured the teaching practices in this domain and, moreover, the design of experimental studies that researchers have been conducting to investigate such practices. In this sense, the mapping results provided us input to build the proposed experimental framework.
- In Chapter 6 we present the results of the experiments that we conducted during this PhD work. The design and lessons learned with the conduction of these experiments also served as a source of information to build the experimental framework.
- In Chapter 7 we provide an overview of our experimental framework for experimental studies on the integration of software testing. We also discuss aspects of the framework creation and show how the experiments conducted by us can be instantiated in the framework.
- Finally, in Chapter 8, we revisit the work conducted in this PhD thesis and highlight its contributions and limitations. We also provide directions of future work and a list of publications that result from this PhD work.

CHAPTER 2

PROGRAMMING EDUCATION

Introductory programming courses compose many undergraduate programs since programming skills are required in many STEM areas. In particular, they are the core of the Computer Science curriculum (ACM/IEEE-CS, 2013). Therefore, the teaching of programming is extensively investigated in the context of computing education (MCCRACKEN *et al.*, 2001; ROBINS; ROUNTREE; ROUNTREE, 2003; LISTER *et al.*, 2004; PEARS *et al.*, 2007; SHEARD *et al.*, 2009; LISTER, 2011; UTTING *et al.*, 2013a; LUXTON-REILLY, 2016; LUXTON-REILLY *et al.*, 2018).

This chapter presents the overview of some aspects of introductory programming in higher education. Using ACM/IEEE curricular guidelines as a base (ACM/IEEE-CS, 2001; ACM/IEEE-CS, 2013), we discuss the recommended content of programming fundamentals in Section 2.1 and the design of programming courses in Section 2.2. In Section 2.3 we discuss several empirical studies with novice programmers, which provide a notion of how students have been responding to programming education in terms of learning outcomes. In Section 2.4 we discuss some teaching approaches that can be used in programming courses, aiming to provide students a learning environment as effective as possible.

2.1 Curriculum

ACM (*Association for Computing Machinery*) and IEEE-CS (*Computer Society of the Institute for Electrical and Electronic Engineers*) have developed **curricular guidelines** to undergraduate computing programs, such as Computer Engineering, Computer Science, Information Systems and Software Engineering[1]. These guidelines include the identification of a **body of knowledge** for Computer Science curricula, hierarchically organized by knowledge *areas*, *units* and *topics*.

---

[1] <http://www.acm.org/education/curricula-recommendations>

Table 1 lists the knowledge areas for the curricular guidelines versions in 2001 and 2013 (ACM/IEEE-CS, 2001; ACM/IEEE-CS, 2013). It is possible to notice the changes between the two versions. The body of knowledge was restructured, even for areas that kept the same name, in order to accompany changes of the area and to better organize the contents which compose the body of knowledge.

Table 1 – Body of knowledge in Computer Science – versions CS2001 and CS2013 (ACM/IEEE-CS, 2001; ACM/IEEE-CS, 2013)

CS2001	**CS2013**
Discrete Structures	Discrete Structures
Programming Fundamentals	**Software Development Fundamentals**
Algorithms and Complexity	Algorithms and Complexity
Architecture and Organization	Architecture and Organization
Operating Systems	Operating Systems
Net-Centric Computing	Networking and Communication
Programming Languages	Programming Languages
Human-Computer Interaction	Human-Computer Interaction
Graphics and Visual Computing	Graphics and Visualization
Intelligent Systems	Intelligent Systems
Information Management	Information Management
Social and Professional Issues	Social Issues and Professional Practice
Software Engineering	Software Engineering
Computational Science and Numerical Methods	Computational Science
	Information Assurance and Security
	Platform-based Development
	Parallel and Distributed Computing
	Systems Fundamentals

The **Programming Fundamentals** area in CS2001 covers the content of introductory programming courses, which are the focus of this PhD project. It is composed by basic programming concepts and an introduction to algorithms and data structures. This area was reformulated and renamed to **Software Development Fundamentals** in CS2013. Table 2 shows the topics that compose both. The content is similar, with some adjustments. For example, the *Recursion* knowledge unit in CS2001 became a topic of the *Fundamental Programming Concepts* unit in CS2013.

Table 3 shows in details the contents of the *Software Development Fundamentals* knowledge area in CS2013. The most significant difference is the inclusion of Software Engineering basic concepts and practices, in the *development methods* unit. Hence, there is a subtle change in the focus of this knowledge area, putting programming in the broader scope of the software development process.

The topics from the *development methods* unit explicitly indicates skills that help students to succeed in programming: identify and find defects in their own code (*defensive programming*, *testing* and *debugging*), restructure the own code to enhance modularization (*simple refactoring*), the use of programming environments etc.

Table 2 – *Programming Fundamentals* area in CS2001 and *Software Development Fundamentals* in CS2013

Units in **Programming Fundamentals** (area in **CS2001**)	Units in **Software Development Fundamentals** (corresponding area in **CS2013**)
Fundamental programming constructs	Fundamental Programming Concepts
Algorithms and problem-solving	Algorithms and Design
Fundamental data structures	Fundamental Data Structures
Recursion	–
Event-driven programming	–
–	**Development Methods**

Table 3 – Contents of Software Development Fundamentals area in CS2013 (ACM/IEEE-CS, 2013)

Units	**Topics**
Algorithms and Design	The concept and properties of algorithms The role of algorithms in the problem-solving process Problem-solving strategies (iterative and recursive mathematical functions , iterative and recursive traversal of data structures, divide-and-conquer strategies) Fundamental design concepts and principles (abstraction , program decomposition, encapsulation and information hiding, separation of behavior and implementation)
Fundamental Programming Concepts	Basic syntax and semantics of a higher-level language Variables and primitive data types (e.g., numbers, characters, booleans) Expressions and assignments Simple I/O including file I/O Conditional and iterative control structures Functions and parameter passing The concept of recursion
Fundamental Data Structures	Arrays Records/structs (heterogeneous aggregates) Strings and string processing Abstract data types and their implementation (stacks, queues, priority queues, sets, maps) References and aliasing Linked lists Strategies for choosing the appropriate data structure
Development Methods	Program comprehension Program correctness (types of errors (syntax, logic, run-time), the concept of a specification, defensive programming, code reviews, testing fundamentals and test-case generation, the role and the use of contracts, including pre- and post-conditions, unit testing) Simple refactoring Modern programming environments (code search, programming using library components and their API) Debugging strategies Documentation and program style

The content of *Programming Fundamentals* or *Software Development Fundamentals* is closely related to introductory computing courses. Even so, it does not mean the introductory sequence must address exactly what is in this area from the body of knowledge. Depending on the emphasis chosen by the instructor to design these courses, the addressed topics may vary. For example, if the instructor choose to use a functional language, she/he can add the coverage of topics on the functional paradigm from the area *Programming Languages*.

2.2 Courses

The sequence of introductory programming courses provides students the needed grounding for advanced computing courses. It may have a variable number of courses, with at least the courses known as **CS1** and **CS2** (ACM/IEEE-CS, 2001). The following examples of introductory sequences may address the same topics with a different sequence design (ACM/IEEE-CS, 2001):

CS101 Programming Fundamentals
CS102 Object-Oriented Paradigm
CS103 Data Structures and Algorithms

CS111 Introduction to Programming
CS112 Data Abstraction

The terms CS1 and CS2 are widely used in the literature, but there is not a consensus of their meaning (HERTZ, 2010). Nevertheless, it is common to consider the introduction to programming concepts course as CS1 and the data structures course as CS2.

There are also disagreements about the division and the sequence of topics that should be covered by these courses (BRUCE, 2005; SCHULTE; BENNEDSEN, 2006). The CS2001 curricular guidelines indicate some models to design the introductory course sequence, which also represent the existing variety of approaches in this sense (ACM/IEEE-CS, 2001):

- ***Imperative-first*:** Topics related to the procedural/imperative paradigm are covered early in this model. It is possible to follow this model with an imperative language, such as C, or even to use an object-oriented one, as Java. In the latter, the imperative aspects of the language are emphasized first and the object-oriented concepts are delayed to another course.

- ***Objects-first*:** This model is characterized by addressing OOP concepts first, which requires the use of an OOP language, such as C++ or Java. Since the first class students learn notions of objects and inheritance. One disadvantage of this approach is that OOP languages are generally more complex, both in terms of syntax and involved concepts. Even so, it is widely adopted due to the increasing importance of OOP in academia and industry.

- ***Functional-first*:** This model implies the adoption of a functional language, such as Scheme (FELLEISEN *et al.*, 2004), during introductory courses. These languages present a simple syntax and allow to express recursion in a more natural way. However, functional languages do not have wide adoption in industry and this can make students less receptive to adopt them.

- ***Breadth-first*:** The idea of this model is to provide to students, at first, an overview of Computer Science areas (Algorithms, Programming Languages, Computer Architecture, Software Engineering, Databases, Operating Systems, Artificial Intelligence, etc) and only after dive into the the teaching of programming. This approach allows students to understand the discipline as a whole first, providing them a better notion to decide whether they want to study it in depth later.

- ***Algorithms-first*:** This model implies the introduction of programming concepts using pseudocode. By delaying the use of a programming language, instructors avoid learning difficulties related to the language syntax. However, this approach can be frustrating to students in the sense that they are not able to see the concrete results of their programs.

- ***Hardware-first*:** All previous models involve teaching programming with a high level language. In this way, students do not get a clear notion about how their programs execute in the computer. The model *hardware-first* has the goal to reduce the "mystery" about how programs are really executed by the machine. This model involves a bottom-up approach with hardware fundamentals (digital circuits, registers, arithmetic units, von Neumann architecture, etc). After learning about the "backstage" of program execution, students learn to program with a high level language. Given its emphasis on hardware, the model may be more adequate to a Computer Engineering curriculum.

The CS2013 curricular guidelines reiterate the existing diversity in these courses, but, instead of indicating models that could be "instantiated" to design the introductory sequence, the document identifies in a more general way the points that may vary when designing the introductory courses (ACM/IEEE-CS, 2013):

- **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 teaching sequence of concepts. Also, this choice can determine the whole underlying model of the introductory sequence (*imperative-first*, *objects-first* and *functional-first*). Naturally, the choice of programming language is also related to the chosen paradigm. There are other important factors, such as industry adoption (e.g. C, C++ and Java) or the simplicity of the syntax (Python) (DAVIES; POLACK-WAHL; ANEWALT, 2011).

- **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 (BRUCE; DANYLUK; MURTAGH, 2010; GROSS, 2011). 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 (MARKHAM; KING, 2010; COWDEN *et al.*, 2012; EDWARDS; ALLEVATO, 2013). 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.

In short, there are a lot of choices to design introductory courses, which involve several tradeoffs that should be considered by instructors (ACM/IEEE-CS, 2013; HERTZ; FORD, 2013). There are still open questions about what topics should be covered in these courses and how they should be taught (KOULOURI; LAURIA; MACREDIE, 2014). The ultimate goal is to improve students' learning outcomes.

2.3 Learning outcomes

Several studies have investigated the learning outcomes of the teaching of programming in higher education. As pointed out by Guzdial (2011), the study conducted by Soloway et al. is one of the first initiatives to investigate the programming performance of students in a CS1 course (SOLOWAY; BONAR; EHRLICH, 1983). The results were disappointing: only 14% of the students were able to develop correct programs to solve the rainfall problem[2].

With similar goals, McCracken *et al.* (2001) performed a study with students from four institutions from different countries (N=217). They asked students to implement three types of calculators: postfix (P1), infix without precedence (P2), and infix with parentheses (P3). Students' programs were evaluated according to the following criteria: execution (*does the program compile and execute without error?*), verification (*does the program handle the inputs correctly?*), validation (*is it the correct type of calculator?*) and style (*does the style of the program conform to local standards, including naming conventions and indentation?*).

Table 4 shows results of students' performance for each programming assignment (P1, P2 and P3). Considering all participants together, the average was 22.9 out of 110 (or 20.81%), with a standard deviation of 25.2. Again, results indicated a low programming performance of students, similar to Soloway's study, and now in a much broader and representative context.

Table 4 – Results of the study conducted by McCracken *et al.* (2001)

	Average (SD)
P1 (N=117)	21.0 (24.2)
P2 (N=77)	24.1 (27.7)
P3 (N=23)	31.0 (20.9)

Figure 1 shows a histogram of student scores for the problem P1. It is possible to notice a bi-modal distribution. Most students performed poorly, which corresponds to the data concentration to the left, and there is a subtle second peak to the right, what indicates a set of students with better performance. This characteristic of bimodality in students' grades has been investigated in the literature (ROBINS, 2010; AHADI; LISTER, 2013).

Lister (2011) mentions that McCracken's findings were a relief for instructors worldwide, because they used to think that problems in their institutions were unusual, specially considering the high failure and drop out rates in the introductory sequence (BEAUBOUEF; MASON, 2005; KINNUNEN; MALMI, 2006; WATSON; LI, 2014; PETERSEN *et al.*, 2016). Then, researchers have focused on investigating why programming is such a difficult subject for students (ROBINS; ROUNTREE; ROUNTREE, 2003).

In this sense, Lahtinen, Ala-Mutka and Jarvinen (2005) conducted a survey to explore such difficulties, both from student and instructor points of view. They asked respondents to assess programming concepts and other related issues from *very easy to learn* (1) to *very difficult* (5). Also, there were questions about teaching approaches, with options from *learning never in that kind of situations* (1) to *learning always* (5), and teaching materials, from *practically useless* (1) to *very useful* (5).

---

[2] "Write a program that will read in integers and output their average. Stop reading when the value 99999 is input." (SOLOWAY; BONAR; EHRLICH, 1983)

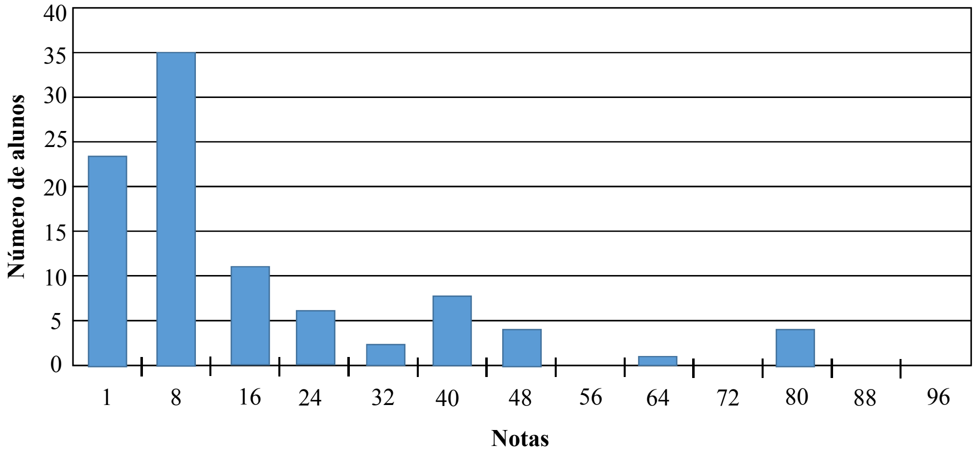


Figure 1 – Distribution of students scores for problem P1 in the study conducted by McCracken *et al.* (2001)

Table 5 presents a summary of their results. For each questionnaire item, there is the average of responses from students and teachers that participated. The issues considered as being most hard are *designing a program to solve a certain task*, *dividing functionality into procedures* and *finding bugs from my own program*, which are related to strategies of designing and debugging the program. These issues are in line with the observations of Robins et al.: the grasp of programming strategies may be more decisive to turn students into effective programmers than programming knowledge (ROBINS; ROUNTREE; ROUNTREE, 2003).

Regarding programming concepts, *recursion*, *pointers/references* and *error handling* are among the ones considered as most difficult. Concerning teaching approaches and materials, it is interesting to observe that questionnaire items referring to practice activities got high grades (towards being very useful/learning always).

There are other follow-up studies to the McCracken group study. McCartney *et al.* (2013) conducted a replication in one institution (N=40) with some changes in the study design. They used a simplified assignment, asking students to implement only the infix calculator (instead of the three types of calculator). They also provided scaffolding code as a starting point for students and allowed them to access other resources such as online documentation. Results were much better: student programming performance had an average of 62%.

Utting *et al.* (2013a) also replicated the McCracken group study, in a multi-institutional context (N=345), with several cohorts (R1, R2, P, T, Q, S and U). They assessed students' programming skill and knowledge, with some study design adjustments as well. The programming assignment was the clock problem, which involves implementing operations with time values (tick, addition, subtration). When compared to the calculators problem, the clock problem is "more in line with object-oriented environments and less algorithmically complex" (UTTING *et al.*, 2013a). Similarly to McCartney *et al.* (2013), they provided scaffolding code to students (a skeleton of the Time class) and, for some groups of students, a test harness (for the Time class).

Table 5 – Results of the survey conducted by Lahtinen, Ala-Mutka and Jarvinen (2005)

Question	Students	Teachers
COURSE CONTENTS		
What kind of issues you feel difficult in learning programming?		
Using program development environment	2,43	2,61
Gaining access to computers/networks	2,11	1,97
Understanding programming structures	2,92	3,27
Learning the programming language syntax	2,75	2,70
Designing a program to solve a certain task	3,12	3,97
Dividing functionality into procedures	3,10	4,06
Finding bugs from my own program	3,28	3,91
Which programming concepts have been difficult for you to learn?		
Variables (lifetime, scope)	2,10	2,41
Selection structures	1,98	2,38
Loop structures	2,09	2,79
Recursion	3,22	4,06
Arrays	2,79	3,24
Pointers, references	3,59	4,44
Parameters	2,60	3,47
Structured data types	2,90	3,45
Abstract data types	3,02	4,06
Input/output handling	2,96	3,75
Error handling	3,33	4,13
Using language libraries	3,04	3,88
LEARNING AND TEACHING PROGRAMMING		
When do you feel that you learn issues about programming?		
In lectures	3,01	3,21
In exercise sessions in small groups	3,44	3,84
In practical sessions	3,77	4,35
While studying alone	3,79	3,42
While working alone on programming coursework	3,98	4,00
What kind of materials have helped/would help you in learning programming?		
Programming course book	3,35	3,30
Lecture notes/copies of transparencies	3,39	3,47
Exercise questions and answers	3,33	3,62
Example programs	4,19	4,24
Still pictures of programming structures	3,15	3,70
Interactive visualizations	3,33	4,07

Table 6 shows students' performance for the clock problem. Considering the results combined, the average score is 2.72 methods working out of 4 (or 68%). The difference between groups with and without test harness revealed interesting findings. Students that implemented the Time class using the test harness had a significant better performance, with an average of 3.26 methods working out of 4 (81.5%). Meanwhile, students without the test harness had an average of only 0.83 methods working (20.75%).

Table 6 – Results of the study conducted by Utting *et al.* (2013a) – skill assessment

group	**test harness?**	**N**	**# methods working (average)**
R1	yes	149	3.04
R2	yes	57	3.86
P	yes	26	3.27
T	yes	38	3.21
Q	no	15	0.80
S	no	40	0.93
U	no	20	0.65
combined	yes	270	3.26
combined	no	75	0.83
combined	all	345	2.72

Using the test harness clearly had a positive effect in students' programming performance. The authors believe that this is due to the scaffolding effect. The tests work as a kind of guidance to what student should implement and test results provide instant and continuous feedback about the implemented program.

Regarding the concept assessment, they used the *Foundational CS1 Assessment Instrument* (**FCS1**), which is composed by multiple choice questions about CS1 concepts (TEW; GUZDIAL, 2011). Table 7 shows students' achieved scores in the FCS1. When combining all the groups, students had an average of 11.35 out of 25, with a standard deviation of 4.711 (42.02%). The correlation between scores in skill and concept assessment was positive (r=0,653).

Table 7 – Results of the study conducted by Utting *et al.* (2013a) – concept assessment

group	**N**	%	σ	median
R1	15	41.73	3.97	11
R2	16	62.27	4.56	17
P	25	60.59	4.23	15
T	57	44.51	4.08	12
Q	17	27.89	3.47	7
S	49	38.17	3.38	10
U	38	28.49	2.68	8

Taking into account the studies conducted by McCracken *et al.* (2001), McCartney *et al.* (2013) and Utting *et al.* (2013a), it is possible to notice the same goals and apparently divergent results. Nevertheless, as pointed out by Hertz and Ford (2013), “just comparing how well students perform may not be accurate as it ignores the many confounding factors that could also have made a difference”. Indeed, the changes in these three studies design, such as use of a less complex programming assignment and providing skeleton code and test harness, are definitely confounding factors which had influence in results.

In short, students presented a significantly better programming performance when instructors designed the programming assignments differently. With a similar reasoning, Luxton-Reilly (2016) raised the following issue. Programming may not be an inherently difficult subject to learn as researchers have believed. Instead, the design of introductory courses as it is may have been establishing unrealistic expectations for novice programmers.

2.4 Teaching approaches

Different approaches can be integrated into the teaching of programming, either to change or to add some aspect in the traditional way (TEW; MCCRACKEN; GUZDIAL, 2005; PEARS *et al.*, 2007; VIHAVAINEN; AIRAKSINEN; WATSON, 2014; KOULOURI; LAURIA; MACREDIE, 2014). The decision of including a given approach can be influenced by several factors, such as students’ background, institutional context, industry demand and even the instructor background (ACM/IEEE-CS, 2013). Again, the ultimate goal is to improve students’ learning outcomes. "Whilst there is no silver bullet, no teaching approach works significantly better than others, a conscious change almost always results in an improvement in pass rates over the existing situation" (VIHAVAINEN; AIRAKSINEN; WATSON, 2014).

In this section, we provide an overview of some teaching approaches which are recurrent in the literature about programming education. Besides the ones discussed here, there are other approaches not restricted to the programming subject, but also often applied in this context: cooperative learning (BECK; CHIZHIK; MCELROY, 2005), active learning (MOURA; HATTUM-JANSSEN, 2011), peer instruction (ZINGARO; PORTER, 2014), flipped classroom (HORTON *et al.*, 2014), POGIL activities (HU; SHEPHERD, 2013), among others.

2.4.1 Software testing

Traditionally, students’ programming skills are strengthened throughout computing courses. They are always practicing in assignments how to develop programs to solve given problems. However, they usually do not practice to the same extent how to validate their solutions. Edwards (2004) highlights some problems related to this issue:

- Students often think that, if the code compiles successfully or executes correctly once or twice, it does not have more errors.

- Instructors often provide feedback after the assignment deadline, which will hardly contribute to students realizing their mistakes during the process of writing the code.
- Typical programming assignments focus on developing students' code writing skills. Other important skills may be overlooked, such as code analysis and comprehension. These kinds of skills would help students to identify and correct errors in their code.

The use of testing practices in programming assignments can help dealing with these problems and benefit students in many ways (BARRIOCANAL *et al.*, 2002; BARBOSA *et al.*, 2003; BARBOSA *et al.*, 2008; JANZEN; SAIEDIAN, 2008; DESAI; JANZEN; CLEMENTS, 2009; SPACCO *et al.*, 2013). The most easily observable benefit is the improvement in the **quality of students' programs**.

Software testing is a topic usually addressed only in advanced computing courses (CHRISTENSEN, 2003; COWLING, 2012). When introducing testing practices earlier in introductory courses, students have more opportunities to learn the pragmatics of testing in programming assignments. Besides, it binds in a more robust way programming and testing activities, improving students' development habits (SPACCO *et al.*, 2013).

2.4.2 Pair programming

Pair programming is a development practice that consists in two programmers developing a program side by side in the same computer (MCDOWELL *et al.*, 2002; NAGAPPAN *et al.*, 2003; HANKS *et al.*, 2011). Each component of the pair has a well-defined role. One person is the *driver*, who controls the mouse and keyboard and is responsible for typing. The other is the *observer*, who actively examines the work done by the driver, looking for errors, thinking about alternatives, searching for resources and considering strategic implications of what has been done (WILLIAMS *et al.*, 2000). After working some time in this arrangement, the components switch roles.

This technique can bring several opportune effects to the teaching of programming (MCDOWELL *et al.*, 2003), such as the improvement of the developed program and a reduction of the time spent to complete an assignment. In addition, students' problem solving ability working in pairs may be better than solo, since the knowledge baggage of the components can be complementary. Pair formation is an important issue when applying the approach in the classroom. In order to obtain the approach benefits, compatibility factors of the students should be considered (SALLEH; MENDES; GRUNDY, 2011).

2.4.3 Visualization

Visualizations are used as educational resources in several areas. The most traditional way to use visualization is to complement course materials and books with figures. Visualization tools apply this same basic idea to dinamically present concepts to students (SORVA; KARAVIRTA; MALMI, 2013).

Visualizations are specially useful in the teaching of programming, since the concepts are inherently abstract (NAPS *et al.*, 2002). Besides, programs and algorithms have a dynamic behavior which is difficult to understand by novices (PEARS *et al.*, 2007). Topics considered as the most difficult (as references, pointers and recursion) are not directly visible in the source code. Therefore, visual representations can make programming concepts more concrete to students.

Some barries to the use of visualizations in the classroom are: difficulties to find quality resources in the desired topics, difficulties in adapting them to a given context, and lack of knowledge about the best way to integrate them into the course activities (SHAFFER *et al.*, 2010; SHAFFER *et al.*, 2011). Ideally, visualizations should be used in an active way by students. If they interact with and answer questions about visualizations, the approach can be more effective in terms of learning outcomes, including an increase in students' motivation (NAPS *et al.*, 2002; EBEL; BEN-ARI, 2006).

2.4.4 Media computation

Programming is a challenging task to non-CS major students, whose focus is not directly related to computing (FORTE; GUZDIAL, 2005). These students may see programming courses as being excessively technical and without a strong relation to real applications (GUZDIAL, 2003). In general, only advanced computing courses reveal the relevance of basic concepts and skills learned in introductory courses. Then non-majors may not have the opportunity to apply programming concepts in a significant context.

Faced with this problem, Guzdial proposed the approach called *Media Computation*, which changes the focus of computing from calculation to communication (GUZDIAL, 2003). The idea is to teach students through the development of programs to manipulate media, such as sound, images, videos and text.

This approach involves adapting course materials to include media computation activities, as the application of image filters, concatenation of sounds, searching Web pages, etc. These kinds of activities are closer to the everyday lives of the students, besides giving an opportunity to use their criativity. In this way, programming can be seen as a communication skill, which is appreciated by students (FORTE; GUZDIAL, 2005).

2.4.5 Robots

Robots can be used as a supporting tool to the teaching of programming. Aiming to increase students' motivation and computing courses retention, the idea is that students write programs to control a robot, while learning programming concepts (MAJOR; KYRIACOU;

BRERETON, 2011). Among the different kinds of robots, *Lego Mindstorms*[3] is the most popular (MCGILL, 2012). Students build a phisical robot with Lego blocks and them program the device that control it. Other example is *Karel the Robot* (BECKER, 2001), which is an environment with a virtual robot programmed by the student.

Since robots have a fun appeal, students' intrinsic motivation and creativity are stimulated (MCGILL, 2012). Summet *et al.* (2009) argued for the effectiveness of personal robots, i.e. each student having her/his own robot. However, personal robots involve a high cost, leading to use only robots available in a laboratory. This limitation can have a negative impact on students' performance, since they would only be able to work on assignments during classes (FAGIN; MERKLE, 2003).

2.5 Final remarks

This chapter presented an overview about the teaching of introductory programming in higher education focusing on recommended content (curriculum), course design, students' learning outcomes and teaching approaches. It is possible to notice the enormous variety of ways to teach programming. Even the recommended topics can vary a lot according to some design choices like paradigm and programming language, as discussed in Section 2.2.

In Section 2.3, the three studies conducted by McCracken *et al.* (2001), McCartney *et al.* (2013) and Utting *et al.* (2013a) have the same goal of assessing students' programming performance in a CS1 course context, but present apparently contradictory results. Yet, looking carefully at the differences among study designs, they present several confounding factors. These factors are related to differences in how the programming assignments were conducted in each study: the problem complexity, with or without scaffolding (skeleton code and test harness), allowed/denied access to other resources, etc). These are variables that could have a systematic effect on students' outcomes.

Another point that is worth mentioning is how software testing stands out in the programming education literature. *Testing fundamentals*, *test case generation* and *unit testing* are recommended programming topics (Section 2.1). *Unit testing* can be integrated into programming courses as a supporting development practice (Section 2.2), which is the same as adopting software testing practices as a teaching approach in these courses (Section 2.4). Finally, results of a multi-institutional study show that testing can improve students' programming performance (study conducted by Utting *et al.* (2013a) in Section 2.3).

---

[3] <http://www.lego.com/mindstorms>

CHAPTER
3

EXPERIMENTATION IN SOFTWARE ENGINEERING

The knowledge produced with experimentation is about the variables involved in a phenomenon. The starting point of an experiment is an educated guess an previous experience about a cause-effect relationship between concepts/constructs of the area. This educated guess is formalized in a **hypothesis**, which is defined in terms of what phenomenon variables should be examined and tested during the experiment. In this way, the role of experimentation is to correspond ideas to reality (JURISTO; MORENO, 2001).

In the Software Engineering area, researchers formulate hypotheses about the development process. Then, they conduct experiments, collecting and analyzing data to verify if the hypotheses are valid. The knowledge about a technology is built from the conduction of several experiments about it (BASILI; SHULL; LANUBILE, 1999). So, in general, experimentation in Software Engineering has the role to evaluate and compare technologies used throughout the development process, always considering the great influence of people that apply them on the observed results.

This chapter presents basic concepts about experimentation in Software Engineering. In Section 3.1 we provide an overview of empirical methods employed in Software Engineering research. In Section 3.2 we focus in one of the methods, controlled experiments, and present which are the basic experiment concepts. In Section 3.3 we show the experimental process, i.e. the steps needed to conduct an experiment. Finally, in Section 3.4 we present domain-specific frameworks, which help researchers to properly design an experiment or even to combine results from existing experiments in a given domain.

3.1 Empirical studies

Empirical studies investigate phenomena around us by means of observation. In Software Engineering the phenomena of interest are related to software development. In general, research in this area involves characterize and evaluate the technologies used during software development, i.e. methods, techniques and tools (SJOBERG; DYBA; JORGENSEN, 2007). Some examples of phenomena investigated in the area are: the use of perspective-based reading techniques in requirements inpection (BASILI *et al.*, 1996), the use of object-oriented technologies in software projects (DELIGIANNIS *et al.*, 2002), the use of testing techniques in the inspection activity (JURISTO; MORENO; VEGAS, 2004), the adoption of agile methods during software development (DYBA; DINGSOYR, 2008), among others.

In this perspective, empirical studies in Software Engineering have been conducted, contributing to build theories in the area and aiming to help in decision-making, since practitioners need to understand and choose adequate technologies to support the development of their projects (SHULL; SINGER; SJOBERG, 2007). The subject of such studies are software practitioners (or students representing them), because the effect of a technology in Software Engineering can only be comprehensily investigated when considering its application by people (BASILI; ZELKOWITZ, 2007a).

There are several methods to conduct empirical studies, such as controlled experiment, case study, survey, ethnographies and action research (EASTERBROOK *et al.*, 2008). Despite that, the steps to acquire knowledge through an empirical study are basically the same (SJOBERG; DYBA; JORGENSEN, 2007): specify a research question, design the study, collect data/evidences, analyze and interpret data. Numerical data implies in the use of **quantitative methods** (statistical analysis) and data in the format of text, images or sounds involves the use of **qualitative methods**. Kitchenham *et al.* (2002) provide important guidelines to conduct all kinds of empirical studies in Software Engineering.

Each empirical method have a set of principles to guide how empirical data should be collected and analyzed. The choice of an empirical method involves several factors, such as the alignment of the method with the research question of interest, available resources, the way researchers want to analyze their data, among others (EASTERBROOK *et al.*, 2008). Three empirical methods stand out in Software Engineering (WOHLIN *et al.*, 2000): survey, case study and experiment.

Surveys are used to characterize people's opinions about a phenomenon. A sample of the population of interest complete a questionnaire (or take an interview). The collected data is the participants' responses, which area analyzed to draw conclusions about the population. In Software Engineering, a survey can be used, for example, to investigate developers' opinions about a technique or tool that has been used for some time.

In a **case study** the phenomenon of interest is monitored while is happens naturally in its context (RUNESON; HOST, 2009). This kind of study can help in the comprehension of how and why the phenomenon happens. Besides, a case study can generate hypotheses to be investigated in future studies. Case studies can be applied to evaluate a technology while it is still in use in a software project. The results are harder to generalize since the researcher has a lack of control in this situation. However, for the same reason, a positive aspect is the high degree of realism.

Controlled experiments require manipulation and control over the investigated phenomenon occurrence. The researcher builds a model of the phenomenon, isolating its influencing factors. Next, she/he establishes which factors should be kept constant and which should vary in order to produce a change in results. The phenomenon occurrence is simulated according to the researcher model.

Experiments usually are executed in a laboratory environment in order to achieve the necessary control. A common experiment design involves the comparison of two situations (for example, using a tecnology and not using it). Then, participants are randomly assigned to groups and generate data to be analyzed. When participant assignment to groups is not ramdon, but the study keep the other characteristics of a controlled experiment, the study is called as **quasi-experiment**.

3.2 Controlled experiments: basic concepts

Experiments are distinguished from other empirical studies mainly because the researcher interaction in the study involves a direct interference in the investigated phenomenon. Thus, the observations are a result of such interference. In order to design this “reality manipulation”, some basic concepts are involved (JURISTO; MORENO, 2001):

- **Experimental units/objects**: The objects are used by the subjects to apply the intervention. Depending on the goal, the experimental unit can be the software project as a whole or any intermediate product obtained during the process. For example, if testing techniques are being compared, the experimental units are the code to which the techniques are applied.

- **Experimental subjects/participants**: Subjects are the people who apply the investigated technologies to the experimental units. Still in the testing techniques example, the subjects are the testers, i.e. people applying the testing techniques.

- **Response/dependent variables**: The response variable represents the result investigated in the experiment. It should be quantitative, i.e. collected data should be mapped to numerical values. It is usually a characteristic of the project, development phase, product or resource, which is measured to verify the effect of the factors.

- **Parameters**: It is any software project characteristic that should be kept constant throughout the experiment. For example: same experience level of the testers, same complexity of the programs under test, etc.

- **Provoked variations/factors/independent variables**: Factors are the project characteristics which are intentionally varied during an experiment and that have an effect in results. Each factor can have several possible alternatives. The experiment aims to examine the influence of these alternatives into the values of the response variables. In the example, the factor is the testing technique used.

- **Alternatives/levels/treatments**: Alternatives are the possible values of a factor. In the example, alternatives can be the functional and structural techniques.

- **Interactions**: If the effect of a factor depends on the value of another, it means that the factors interact. Interactions require a specific type of experimental design: the factorial design.

- **Undesired variations/blocking variables**: It is not always possible to keep constant all characteristics that are not of interest. These undesired variations are known as blocking variables. The block design accommodates this kind of variation and provides a correct way to evaluate its influences.

- **Replications**: Replication is the repetition of an experiment. The purpose is to verify previous observed results (GOMEZ; JURISTO; VEGAS, 2010). If a replication in another context provides results which are consistent with the original experiment, the confidence in the hypothesis increases. A replication does not have to be identical to the original experiment, even because it is impossible to find identical objects and subject and put them through the same conditions. Therefore, variations are expected and it is important to know how to characterize them (SHULL *et al.*, 2008).

- **Experimental error**: Even if an experiment was repeated with approximately the same conditions, obtained results will never be completely identical. These result variations are called experimental error.

3.3 Experimental process

An experiment involves a sequence of steps to plan and execute it. The purpose of an experimental process is to describe such steps, serving as a checklist (what) with guidelines (how) of experimental activities that a researcher need to perform in order to conduct an experiment (WOHLIN *et al.*, 2012). Both Wohlin *et al.* (2012) and Juristo and Moreno (2001) present descriptions of the experimental process, as outlined in figures 2 and 3.

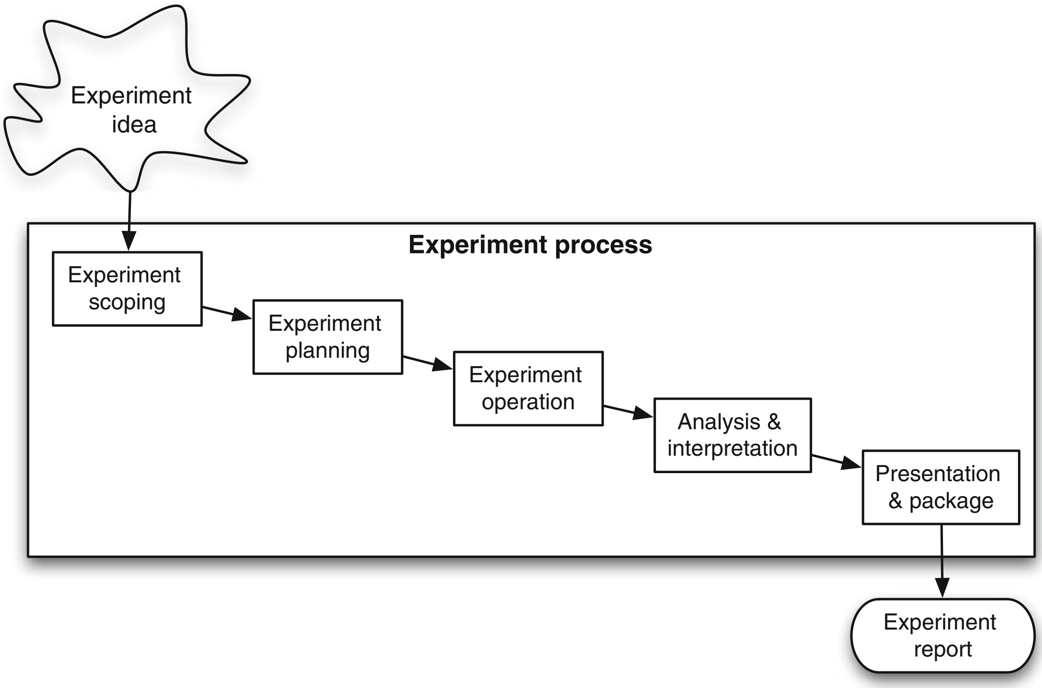


Figure 2 – Experimental process of Wohlin *et al.* (2012)

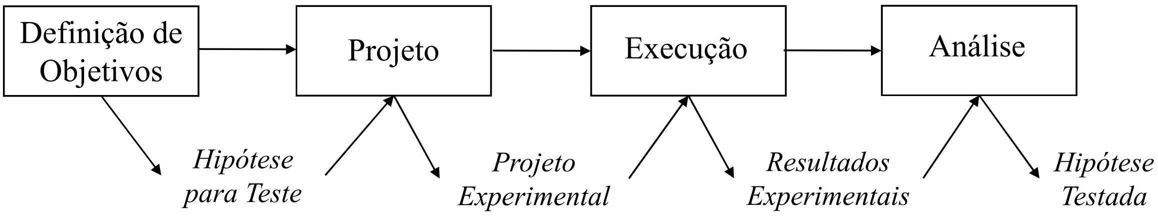


Figure 3 – Experimental process of Juristo and Moreno (2001)

Both processes are structured in a similar way. The main difference is that Wohlin *et al.* (2012) also consider the phase of Presentation and Packaging, which refers to the experiment documentation. Although they do not include a separate phase to discuss this issue, Juristo and Moreno (2001) also present documentation guidelines, emphasizing what elements an experiment report should have.

In this section we describe the experimental process according to the structure of Wohlin *et al.* (2012), using many concepts and guidelines provided by Juristo and Moreno (2001). In this way, the positive aspects of both works are considered. In the **scoping** phase the experiment goal is defined. In the **planning** phase, the hypotheses and the experimental design are formulated. Then, experiment is actually executed in the **operation** phase. **Presentation and packaging** refers to the preparation of the experiment report.

3.3.1 Scoping

The researcher places the experiment in the research area during the scoping phase. As Figure 4 shows, this phase involves expressing the researcher initial idea in terms of a goal, using concepts of the area.

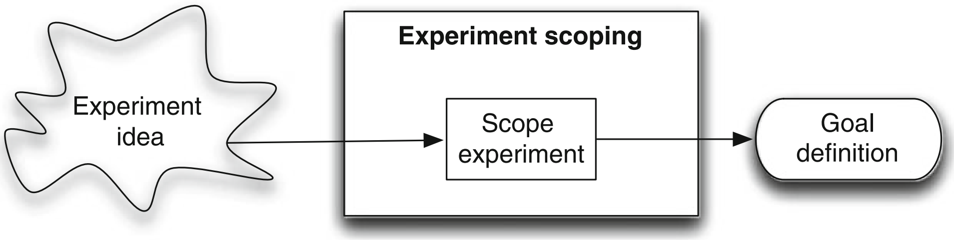


Figure 4 – Scoping phase – experimental process (WOHLIN *et al.*, 2012)

The experiment goal can be instantiated from the goal template of the GQM model (BASILI; CALDIERA; ROMBACH, 1994):

Analyze <*Object(s) of study*>
for the purpose of <*Purpose*>
with respect to their <*Quality focus*>
from the point of view of the <*Perspective*>
in the context of <*Context*>

The values from Table 8 can be used as a starting point to define each template element:

- The **object of study** is the investigated entity.
- The **purpose** reveals the investigation nature.
- The **quality focus** is the effect of the object of study that will be investigated in the study.
- The **perspective** indicates the point of view from which the experiment is designed, i.e. for whom the experiment results will be useful.
- The **context** is characterized by the participants (subjects) and software artifacts (objects).

3.3.2 Planning

During the planning phase the experiment goal is refined towards defining an action plan for the researcher. Two elements should be defined in this phase: the **hypothesis** and the **experimental design**. The activities of this phase are depicted in Figure 5 and described in the next subsections.

Object of study	Purpose	Quality focus	Perspective	Context
Product	Characterize	Effectiveness	Developer	Subjects
Process	Monitor	Cost	Modifier	Objects
Model	Evaluate	Reliability	Maintainer	
Metric	Predict	Maintainability	Project manager	
Theory	Control	Portability	Corporate manager	
	Change		Customer	
			User	
			Researcher	

Table 8 – Goal definition values (WOHLIN *et al.*, 2012)

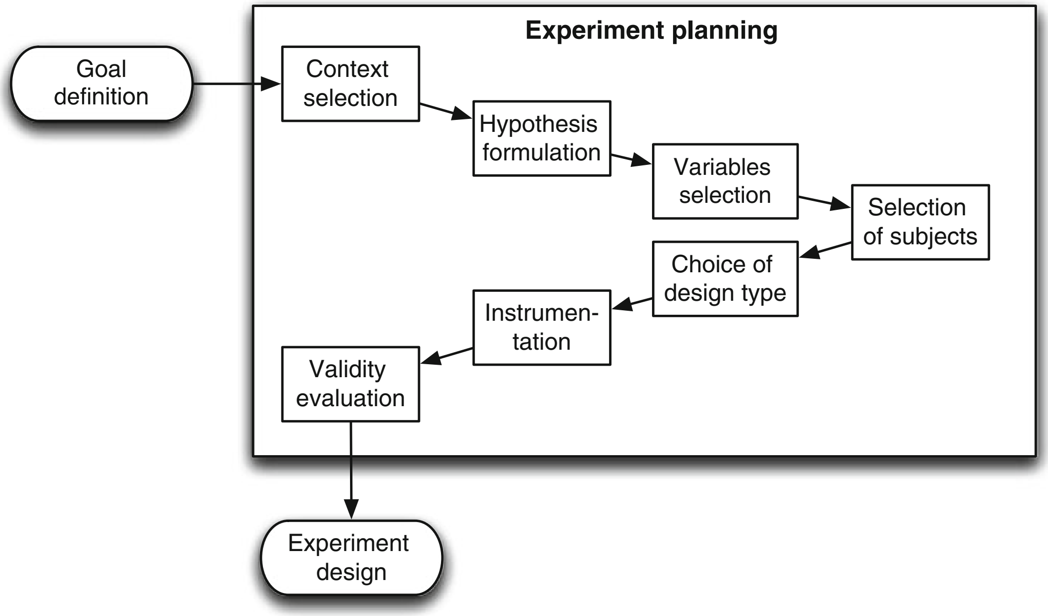


Figure 5 – Planning phase – experimental process (WOHLIN *et al.*, 2012)

Context selection

The experiment context is refined during this activity. The context refers to the environment where the experiment will be executed, which is characterized by experimental subjects and objects. Ideally, the execution should happen in conditions similar to a real software project, involving real software problems and practitioners of the area (SJOBERG *et al.*, 2003). However, given the associated high cost, experiments in Software Engineering are usually conducted in an academic context, with undergraduate and graduate students as subjects (CARVER *et al.*, 2003; CARVER *et al.*, 2010; DEKHANE; PRICE, 2014).

Hypothesis formulation

Hypothesis formulation involves expressing it in terms of a cause-effect relationship, which should be testable, i.e. mapped to variables that can be measured. Strictly speaking, an experiment cannot prove that a hypothesis is true, but only fail to prove it is false[1]. For that reason, two kinds of hypothesis are formulated in an experiment:

- A **null hypothesis** (H_0), which denies the researcher's guess. If accepted, it means that the collected data does not indicate the expected pattern of relationship between alternatives and results.
- An **alternative hypothesis** (H_a ou H_1), which consists in the researcher's guess. It can be considered if the null hypothesis has been rejected.

Variable selection

The **variables** in an experiment represent the cause-effect relationship expressed in the hypothesis. As depicted in Figure 6, the cause variables are the *input variables*, because they represent the influences of the phenomenon under investigation. Similarly, the *output variables* represent the effect expressed in the hypothesis. In other words, they represent the effect of the input variables. Variable selection consists in determining which will be the influences and the effects examined during the experiment.

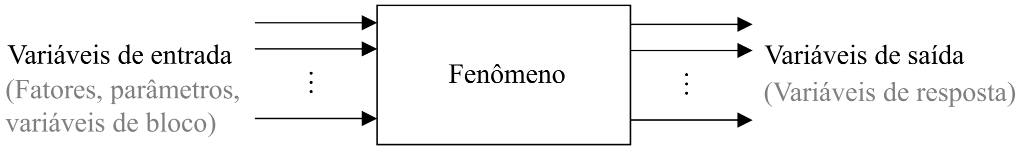


Figure 6 – Input and output variables in an experiment

The researcher have to consider the influences to which a software project (or the project part under investigation) is subject in order to identify the input variables. A software project depends on many factors (involved people, conducted activities, methods used etc). Given the difficulty to analyze several factors in a single experiment, some factors of interest are selected, aiming to isolate their effects from the remaining factors. Figures 7 and 8 show internal and external influences of a software project, which can be considered as candidates to the input variables selection.

The *factors* are the influences of interest which are manipulated during the experiment. The *parameters* are the influences not of interest, which should be configured with constant values throughout the experiment execution. In case that is not possible to keep them constant, they should be considered as *blocking variables*.

The manipulation consists in assigning factors predetermined values during the execution, i.e. their *alternatives/treatments/levels*. Experiments with just one factor and two alternatives

---

[1] This represents the falsifiability scientific principle: if a hypothesis is falsifiable, it should be possible to prove that it is false.

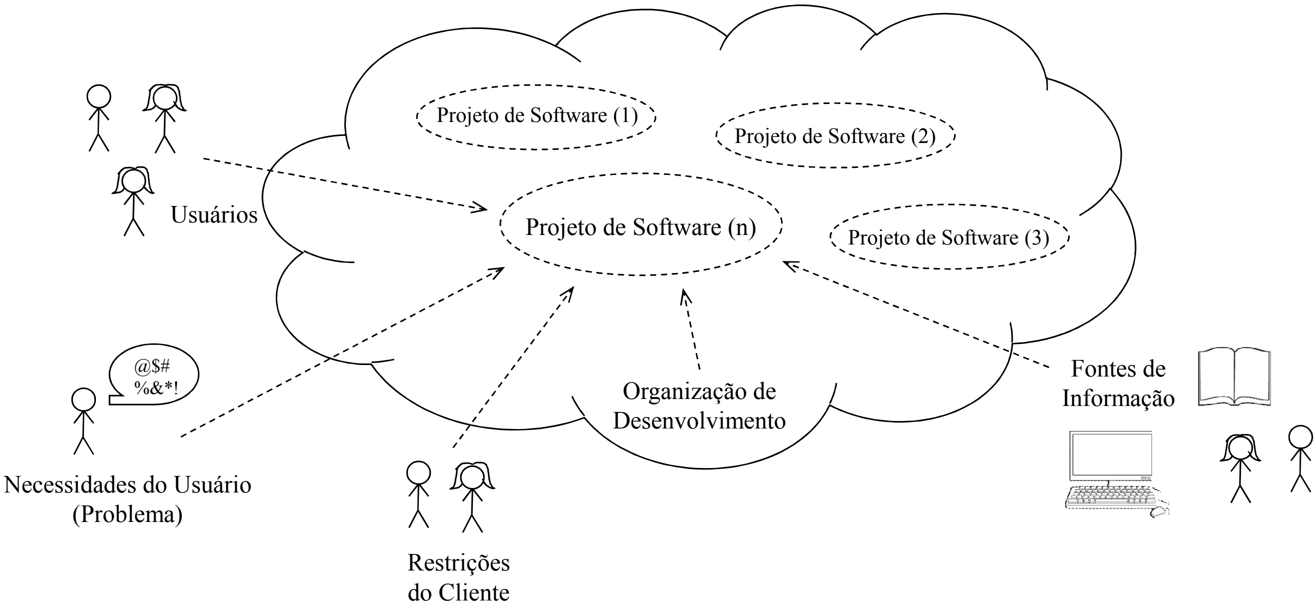


Figure 7 – Influencias externas de um projeto de software (JURISTO; MORENO, 2001)

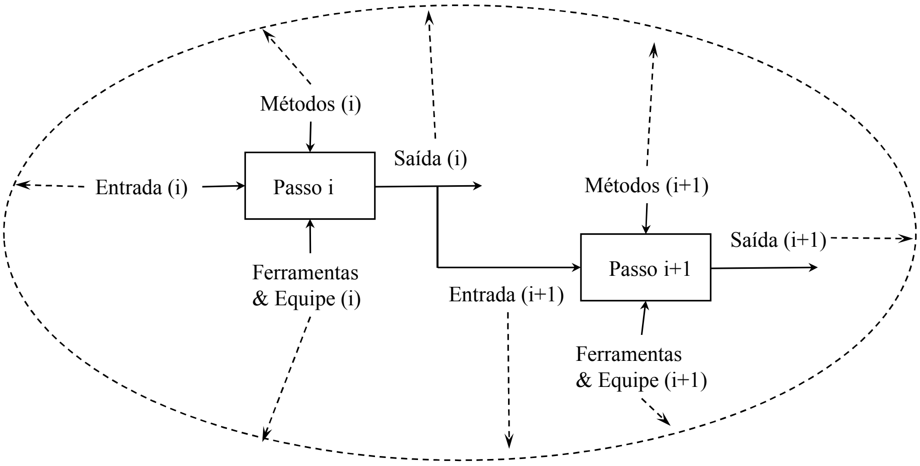


Figure 8 – Influencias internas de um projeto de software (JURISTO; MORENO, 2001)

are fairly common. For example, the factor can be tool support in a given development activity and the alternatives with and without the tool. Table 9 shows examples of factor and respective values taht can be used as alternatives in experiments. These examples in particular re related to internal influences depicted in Figure 8. But external influences could be used as factors as well.

The **response variables** hold values of experiment results. They consist in characteristics of the development process, methods or tools, the personnel, or the intermediate products obtained throughout the process. Table 10 shows examples of response variables for each of these elements.

In addition to selecting which are the response variables of the experiment, it is also necessary to determine how these variables should be measured. The **Goal Question Metric (GQM) model** (BASILI; CALDIERA; ROMBACH, 1994) can be used to help the identification

Table 9 – Examples of factors and respective values in Software Engineering – adapted from Juristo and Moreno (2001)

Variables (factors)	**Values**
METHODS AND TOOLS	
Method	Name of the methods used in a given activity
Tool	Name of the tools used in a given activity
PERSONNEL	
Size	Number of people
Structuredness	Number per position
Assignment	Tasks to be performed by members
Level of communication	High, medium, low
Level integration	High, medium, low
Level of excellence	Average, high
Background experience in domain	None, some, very experienced
Background experience in application type	None, some, very experienced
Knowledge of SE	None, some, very knowledgeable
Experience in the software process	None, some, very experienced
Practical experience in SE	None, some, very experienced
Experience in tools/methods	None, some, very experienced
Experience in position	None, some, very experienced
PRODUCT *(activities input/output)*	
Document legibility	None, little, a lot
Size	Large, medium, small
Software architecture	Object-oriented organization, multi-layer organization, repositories, etc
Type of module	Model calculations, user I/O, control, error processing, help messages processing, moving data around, comments, data declaration

Table 10 – Examples of response variables in Software Engineering – adapted from Juristo and Moreno (2001)

Development process	Schedule deviation, budget deviation, process compliance
Methods	Efficiency, usability, adaptability
Resources	Productivity
Products	Reliability, portability, usability of the final product, maintainability, design correctness, level of code coverage

of response variables of the experiment. As depicted in Figure 9, the model links variables measured during the experiment with its goals. Hence, the model supports the researcher since goal definition (with the goal template discussed in Section 3.3.1), which are then directed towards the definition of research questions and, finally, the identification of metrics that allow to answer the questions.

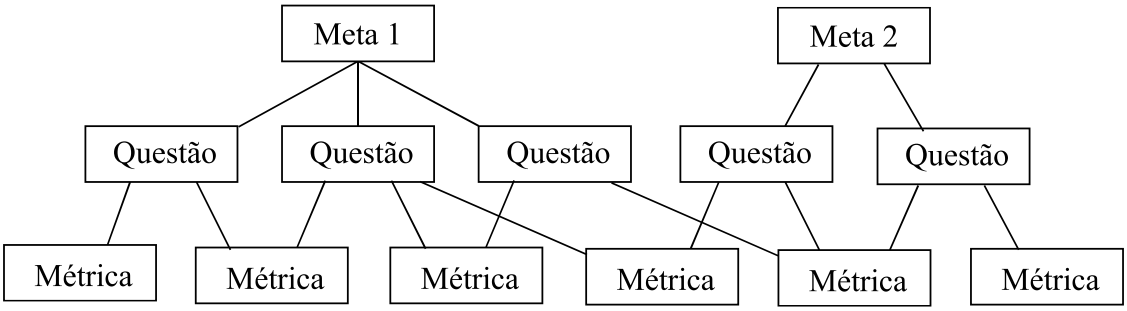


Figure 9 – Goal Question Metric model (BASILI; CALDIERA; ROMBACH, 1994)

Subject selection

Subject selection involves a sampling of the population of interest. The selected sample influences in the generalization of results and, therefore, subject selection should be representative of the population of interest. In this sense, *sampling techniques* (random sampling, if possible) should be considered and the *size sample* should be big enough to deal with population variability and increase the power of the statistical test.

Experimental design

In the **experimental design** "real world adjustment" to the experiment, i.e. the researcher designs the model of the phenomenon that should executed during the experiment: the arrangement of variables, subjects and objects. In Figure 10 some commonly used exerimental designs are outlined, which can be chosen according to the configuration of the experiment variables. These examples provide the basic structure of an experimental design, to which the assignments of subjects and objects can be done later.

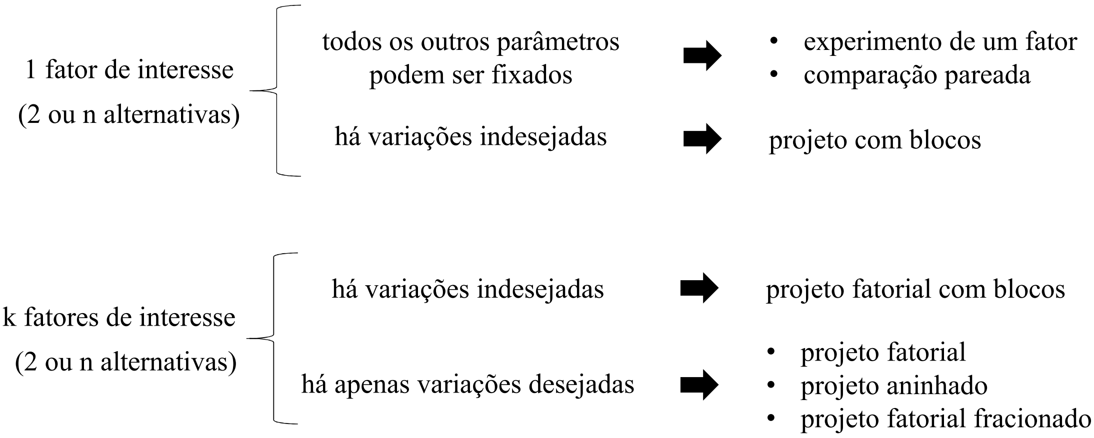


Figure 10 – Diferentes projetos experimentais (JURISTO; MORENO, 2001)

The role of the experimental design structure is to determine how should be the combination of factor alternatives. For example, if there are two factors A and B with alternatives A1, A2, B1 and B2, in a factorial design the alternatives are combined in the following way: A1B1, A1B2, A2B1, A2B2.

However, to complete the experimental design of the example, it is still necessary to assign these four combinations in a ramdon way to objects and subjects. The assignment of values to the experimental design structure should be done in a ramdon way, because a systematic assignment (or the reason behind it) could create sources of undesired variations in the experiment.

Instrumentation

During instrumentation all experiment materials/instruments are prepared. This activity provides means to execute and monitor the experiment. The instruments of an experiment are: the objects, subject guidelines and data collection forms.

The **objects** chosen in the context selection activity are now adjusted to the tasks that subjects will complete. For example, if the experiment investigates the perspective-based reading technique, the requirement document should have seeded defects, once the subjects are expected to find defects in this kind of document using the reading technique.

Subject guidelines are instructions about the tasks, with explanations about how the investigated technology should be used/applied. It may be a process description or a checklist given to the subjects during execution, for example. It is necessary to analyze the need to perform a training with the subjects before performing the actual tasks.

Data collection forms are means to carry out measurements of the response variables. Data collection may be through interviews or subjects themselves recording data generated during the tasks. Anyway, forms should be designed in such a way that it does not impose additional difficulties to whom is registering the data.

Validity evaluation

The validity evaluation is performed before experiment execution, aiming to identify validity problems and adjust the experimental design accordingly. The purpose is to design the experiment for obtaining valid results.

There are four types of validity in an experiment: construction, internal, conclusion and external. Evaluating each one of these types implies in evaluating different stages of the experiment conduction, as outlined in Table 11. Each validity type reflects one of the stages needed to transform the researcher's initial idea into an experiment and then into generalized conclusions back to theory.

The purpose of this activity is to identify **threats to validity** of the results in each of these stages. Table 12 presents a set of possible threats to validity, which can be used as a checklist by

Table 11 – Validity types

Experiment stage	Validity kind
Transform cause and effect constructs from theory into observable experiment variables	construct validity
Determine what variables will be examined (which will be manipulated and which will be kept constant) and what will be their observed effects	internal validity
A statistical test is applied to the measured effects of the alternatives	conclusion validity
The conclusions obtained from the experiment sample are generalized to the population of interest	external validity

the researcher (further details can be found in Wohlin *et al.* (2012)). In general, for each type of validity the researcher should verify the work done:

- Evaluating the **construct validity** implies in checking whether the theory is well mapped into the model used by the experiment, i.e. the alternatives should represent well the cause construct and the results should represent well the effect construct.
- **Internal validity** refers to the causal relation between variables selected for the experiment. The threats to this type of validity are the **confounding factors**, which are unknown factors influencing results, but not identified or controlled by the researcher.
- **Conclusion validity** is related to the conclusions obtained through the statistical analysis of results. Threats of this type involves choice of the statistical test, sample size, performed measures etc.
- **External validity** refers to the generalization of conclusions to the population of interest, which is a more general context than the one imposed by experiment execution conditions. Threats of this kind are related to the choice of subjects and objects.

3.3.3 Operation

During the operation phase the experimental design is put into practice, i.e. the experiment is actually executed. Figure 11 outlines the activities of this phase, which takes the experimental design as input and generates the collected data as output.

The **preparation** activity refers to the last settings that enable the experiment execution. The material kit is prepared, including training materials, objects, guidelines, consent forms, collection forms etc. A **pilot study** is an interesting way to validate these materials (MENDONCA *et al.*, 2006).

Next, in the **execution** activity subjects perform the tasks, applying the alternatives and generating experimental data. The duration of this activity varies a lot: it could be a short session

Table 12 – Threats to validity (WOHLIN *et al.*, 2012))

Conclusion validity	**Internal vadility**
Low statistical power Violated assumption of statistical tests Fishing and the error rate Reliability of measures Reliability of treatment implementation Random irrelevancies in experimental setting Random heterogeneity of subjects	History Maturation Testing Instrumentation Statistical regression Selection Mortality Ambiguity about direction of causal influence Interactions with selection Diffusion of imitation of treatments Compensatory equalization of treatments Compensatory rivalry Resentful demoralization
Construct validity	**External validity**
Inadequate preoperational explication of constructs Mono-operation bias Mono-method bias Confounding constructs and levels of constructs Interaction of different treatments Interaction of testing and treatment Restricted generalizability across constructs Hypothesis guessing Evaluation apprehension Experimenter expectancies	Interaction of selection and treatment Interaction of setting and treatment Interaction of history and treatment

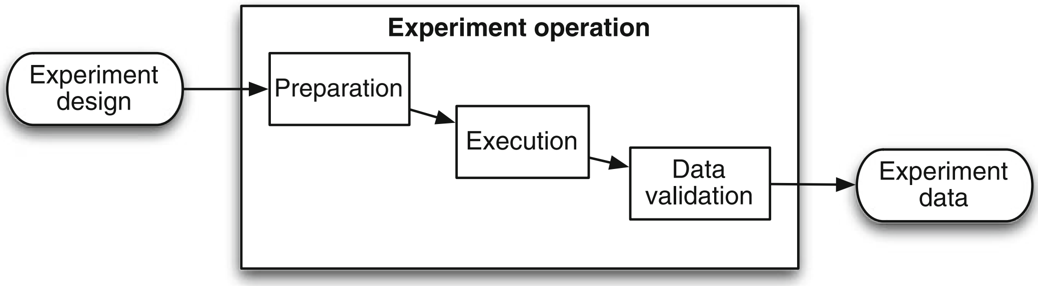


Figure 11 – Operation phase – experimental process (WOHLIN *et al.*, 2012)

or over a project that takes months. The important principle here is to control the execution (as far as possible), in order to assure the desired manipulation for the experiment and avoid undesired variations.

Finally, researcher performs **data validation**, aiming to verify whether data was generated and collected correctly. This activity involves check if participants in fact understood what they were supposed to do or if there was a misunderstanding, which could invalidate the collected data. This verification could be done by showing subjects the data and asking if they agree with the obtained results.

3.3.4 Analysis and interpretation

The analysis phase has the role to deal with data collected from the sample (part) and to make the inference about the population (whole). Therefore, this activity is directly linked to Statistics, which is the area that supports the process of answering questions and making decisions through data analysis. The analysis activities are outlined in Figure 12.

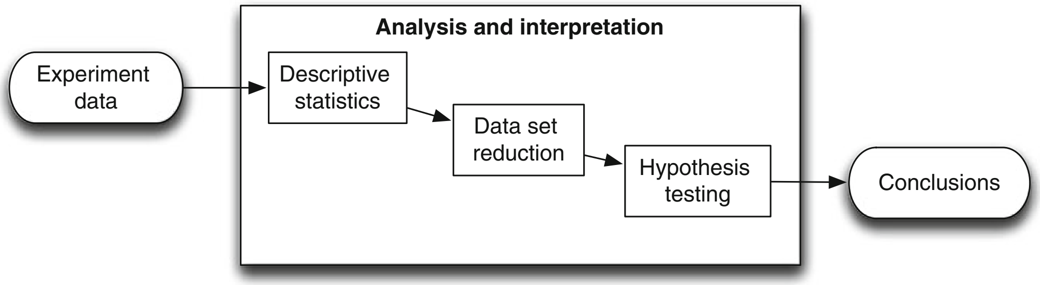


Figure 12 – Analysis phase – experimental process (WOHLIN *et al.*, 2012)

In the first activity, the researcher represents the data with **descriptive statistics** using some measures, such the ones listed in Table 13, and graphically using plots, e.g. histograms, pie charts, box plots. Note that the measures depend on the variable scale type. This data overview allows to detect *outliers*, i.e. data points that differ greatly from others. The researcher should analyze the reasons behind outliers to perform the **data set reduction** if necessary, i.e. to decide whether they should be deleted or not.

Table 13 – Measures of descriptive statistics by scale type (WOHLIN *et al.*, 2012)

	Measure of		
Scale type	**central tendency**	**dispersion**	**dependency**
nominal	mode	frequency	
ordinal	median, percentile	interval of variation	correlation coefficient (Spearman, Kendall)
interval	mean, variance and range	standard deviation	correlation coefficient (Pearson)
ratio	geometric mean	coefficient of variation	

The last activity is the **hypothesis testing**. The hypothesis is tested with the sample results. If they diverge considerably from expected relationship pattern, hypothesis can be rejected, or at least not accepted in the face of the obtained evidence. As indicated by Figure 13 and Tabela 14, the choice of statistical test depends on the type of experimental design, result measurements scale and data distribution.

If the null hypothesis was rejected, it is possible to consider the alternative hypotheses with the obtained data, if results are considered **valid**, considering the experiment validity evaluation. The results generalizationshould be done to environments similar to the one configured in the experiment.

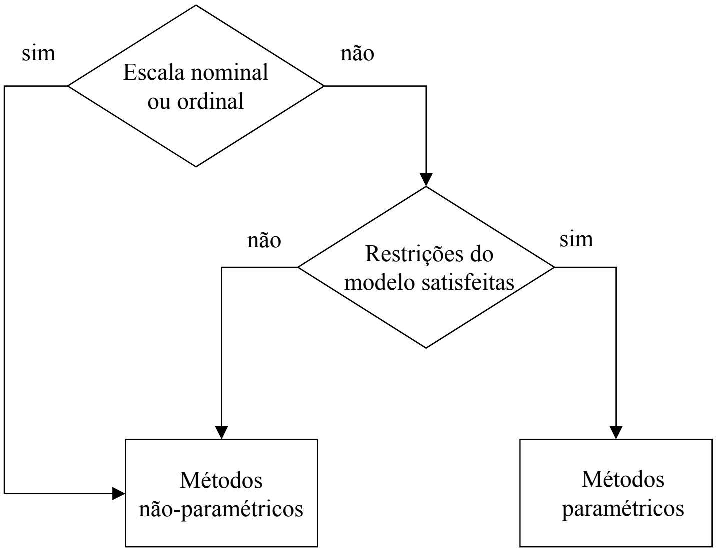


Figure 13 – Criteria to choose between parametric or non-parametric tests (JURISTO; MORENO, 2001)

Table 14 – Overview of parametric and non-parametric tests for different designs (WOHLIN *et al.*, 2012)

Design	**Parametric**	**Non-parametric**
One factor, one treatment		Chi-2, Binomial test
One factor, two treatments, completely randomized design	t-test, F-test	Mann-Whitney Chi-2
One factor, two treatments, paired comparison	Paired t-test	Wilcoxon, Sign test
One factor, more than two treatments	ANOVA	Kruskal-Wallis Chi-2
More than one factor	ANOVA	

3.3.5 Presentation and package

The presentation phase refers to the documentation of the experimental design, adopted procedures, results and obtained conclusions. As indicated in Figure 14, the output is the experiment report. this phase is related to important principles of good scientific practices, such as transparency and reproducibility.

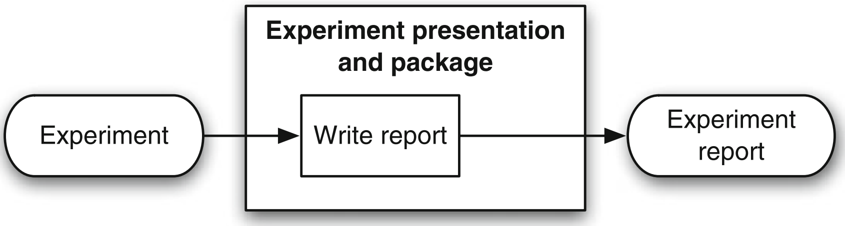


Figure 14 – Packaging phase – experimental process (WOHLIN *et al.*, 2012)

Vegas *et al.* (2006) identify two ways to disseminate information about an experiment:

(i) experiment documentation and (ii) communication (or interaction) among researchers. The documentation is related to this phase. According to Solari and Vegas (2006), experiment documentation can be done by means of a research paper or a laboratory package (also known as experimental package or replication package). Jedlitschka, Ciolkowski and Pfahl (2008) provide guidelines on experiment reporting.

3.4 Experimental frameworks

Software development involves different social, technological and organizational/institutional aspects (BASILI; ZELKOWITZ, 2007b). Thus, an isolated experiment investigates only a limited configuration of such aspects, i.e. a small part the composes the whole of a research domain. Therefore, researchers should be able to combine studies in a given domain in order to create an evidentiary base (WILLIAMS; LAYMAN; ABRAHAMSSON, 2005).

However, "mismatches between empirical studies are the key type of problems that hinder the combined use of independently developed studies" (SHULL *et al.*, 2005). In order to deal with these mismatches, a common framework can serve as a frame of reference, by making explicit the different models used in each study. Also, a framework can provide "a focus for future studies, i.e., to help determine the important attributes of the models used in an experiment and which should be held constant and which should be varied in future studies" (BASILI; SHULL; LANUBILE, 1999).

We identified several proposals of frameworks in the SE literature, aiming to incorporate domain models. Authors use different names to refer to this kind of frameworks: *organizational framework* (BASILI; SHULL; LANUBILE, 1999), *research framework* (GALLIS; ARISHOLM; DYBA, 2003) and *evaluation framework* (WILLIAMS *et al.*, 2004; WILLIAMS; LAYMAN; ABRAHAMSSON, 2005; MORRISON, 2015). An ontology of the research domain can also be used with the same purpose (KITCHENHAM *et al.*, 1999).

In general, experimental frameworks provide the basic structuring of experiments in a given domain. Each experimental activity indicates *what* the researcher have to do, e.g scope experiment, context selection, hypothesis formulation, variables selection and so on. However, in order to complete them, the researcher have to define several domain-specific elements and generate the "structure" of the experiment.

Figure 15 present the domain-specific elements that should be defined in the experiment scoping and planning phases. Note that they revolve around the experiment variables. The colors in the figure suggest their relations: factors are characteristics of the object of study, response variables are the quality focus, and the context is characterized by the experimental subjects and objects. Also, a hypothesis is a testable statement, containing an educated guess of the researcher about the relationship between factors and response variables.

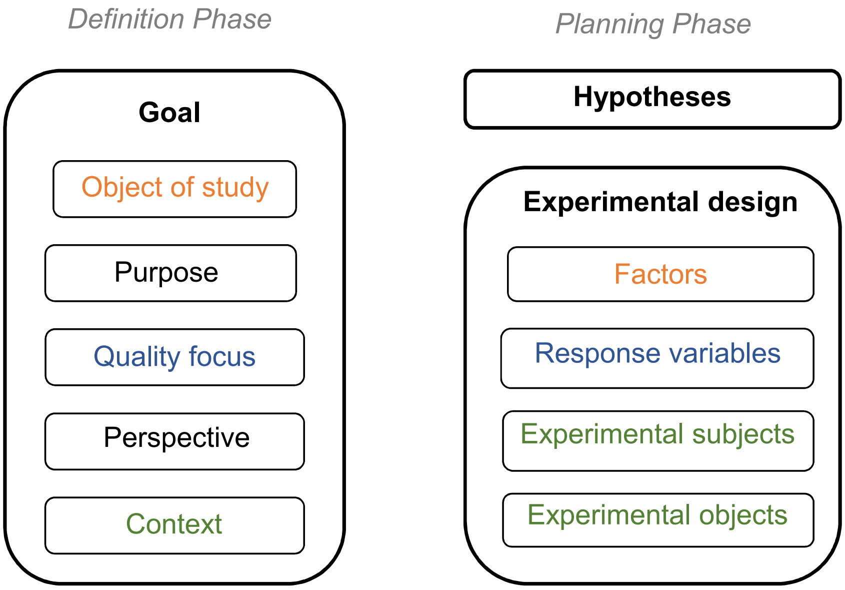


Figure 15 – Domain-specific experiment elements defined in the scoping and planning phases

Therefore, the variable selection is crucial in the experimental process. A poor selection of variables in an experiment, like overlooking factors of a phenomenon (WOHLIN *et al.*, 2012), can hinder the researcher in the effort of relating collected data back to the concepts in theory (EASTERBROOK *et al.*, 2008).

3.4.1 Framework on software reading techniques

Basili, Shull and Lanubile (1999) presented a framework for software reading techniques experiments. The authors considered a set of experiments conducted by themselves on different reading techniques (defect-based reading, perspective-based reading, use-based reading and scope-based reading). It is a general structure designed to accommodate this set of related studies, also known as family of studies.

Their framework consist of the experiment goal given in Table 15. The goal is generic enough to instantiate studies for the different kinds of reading techniques. In order to describe an experiment according to this structure, the generic values (*process*, *effectiveness* and *document*) should be exchanged for specific values, which are given by the models in figures 16, 17 and 18.

Analyze	***processes***	<object of study>
with the purpose to	***evaluate***	<purpose>
with respect to	***effectiveness in a product***	<quality focus>
from the point of view of the	***researcher***	<perspective>
in the context of	***<a set of context variables>***	<context>

Table 15 – Instantiable goal from the organizational framework of Basili, Shull and Lanubile (1999)

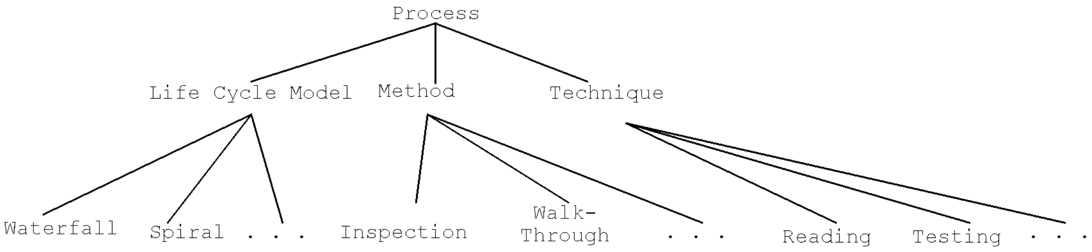


Figure 16 – Possible values for software processes (BASILI; SHULL; LANUBILE, 1999)

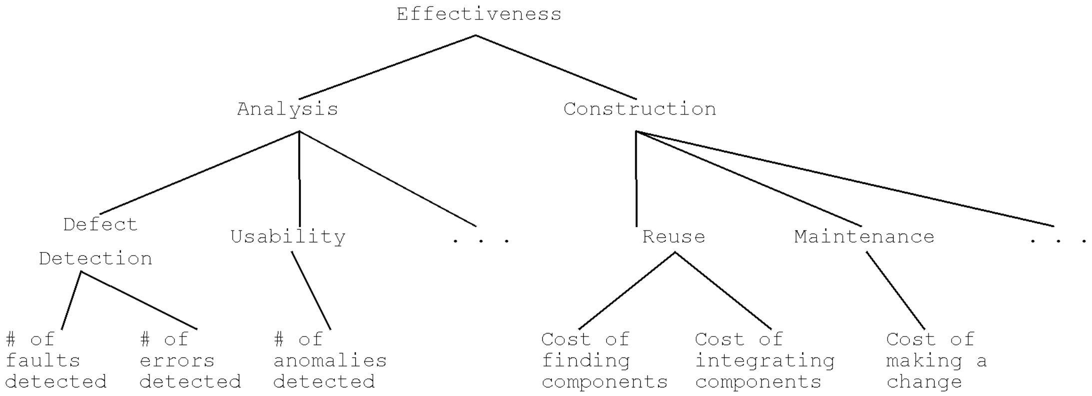


Figure 17 – Possible values for describing the effectiveness of software processes (BASILI; SHULL; LANUBILE, 1999)

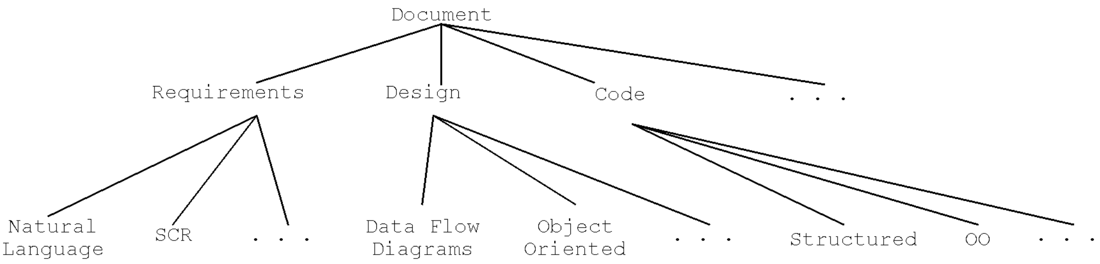


Figure 18 – Possible values for describing software documents (BASILI; SHULL; LANUBILE, 1999)

For example, given an experiment to investigate the perspective-based reading technique (BASILI *et al.*, 1996), its goal could be described as a framework instance: analyze *reading techniques* to evaluate the *ability to detect defects* on *natural language requirements documents* (BASILI; SHULL; LANUBILE, 1999). The values of this instance are highlighted in figures 16, 17 and 18.

3.4.2 Framework on pair programming

Gallis, Arisholm and Dyba (2003) presented a framework for pair programming studies. The authors considered the set of studies listed on Table 16, besides their own studies being conducted at the time. They created a structure of variables that can be selected in an experiment on pair programming, as depicted in Figure 19.

The authors advocate that studies on this domain had apparently contradictory results due to choices in experimental design. For example, the studies that investigate quality as a dependent variable use different metrics: readability and functionality, number of passed test cases, and number of lines of code and number of resubmissions due to defects in code. When variables are operationalized differently like that, it is difficult to compare results.

Table 16 – An overview of existing empirical studies on pair programming (GALLIS; ARISHOLM; DYBA, 2003)

Author(s)	Type of study	Subjects	N	Task	Duration	Independent variable(s)	Main dependent variables (metrics)
(NOSEK, 1998)	Experiment	Prof.	15	Unknown application domain (database script)	45 minutes	Individuals (5) versus pairs (5)	Quality (readability and functionality), Programmers morale (qualitative assessment)
(WILLIAMS *et al.*, 2000; WILLIAMS, 2000)	Experiment	Stud.	41	Four programming assignments	Six weeks	PSP (13) versus CSP (14 pairs)	Time to complete the assignments (number of hours from start to finish), Cost (number of programmer hours), Quality (number of passed test cases)
(NAWROCKI; WOJCIECHOWSKI, 2001)	Experiment	Stud.	21	Four programs proposed by W. Humphrey	N/A	PSP (6), XP with PP (5 pairs) e XP with individual progr. (5)	Time (number of hours from start to finish – elapsed time), Quality (number of lines of code and number of resubmissions due to defects in code)
(MCDOWELL *et al.*, 2002)	Experiment	Stud.	313	Course assignments	Two academic semesters	Individual (141) versus pair programming (86 pairs)	Quality – score on programming assignment (functionality and readability), Learning effect (score on final exam)
(MULLER; TICHY, 2001)	Case study	Stud.	12	Software tasks	11 weeks	Evaluation of XP (including PP) to gather experience with the process	Information and knowledge transfer (qualitative assessment), Morale (qualitative assessment)
(GALLIS; ARISHOLM; DYBA, 2002)	Case study	Prof.	4	Project coding tasks	Project estimate: 5 months	Partner programming (1 pair) versus PP (1 pair)	Information and knowledge transfer (qualitative assessment), Programmers morale (qualitative assessment)

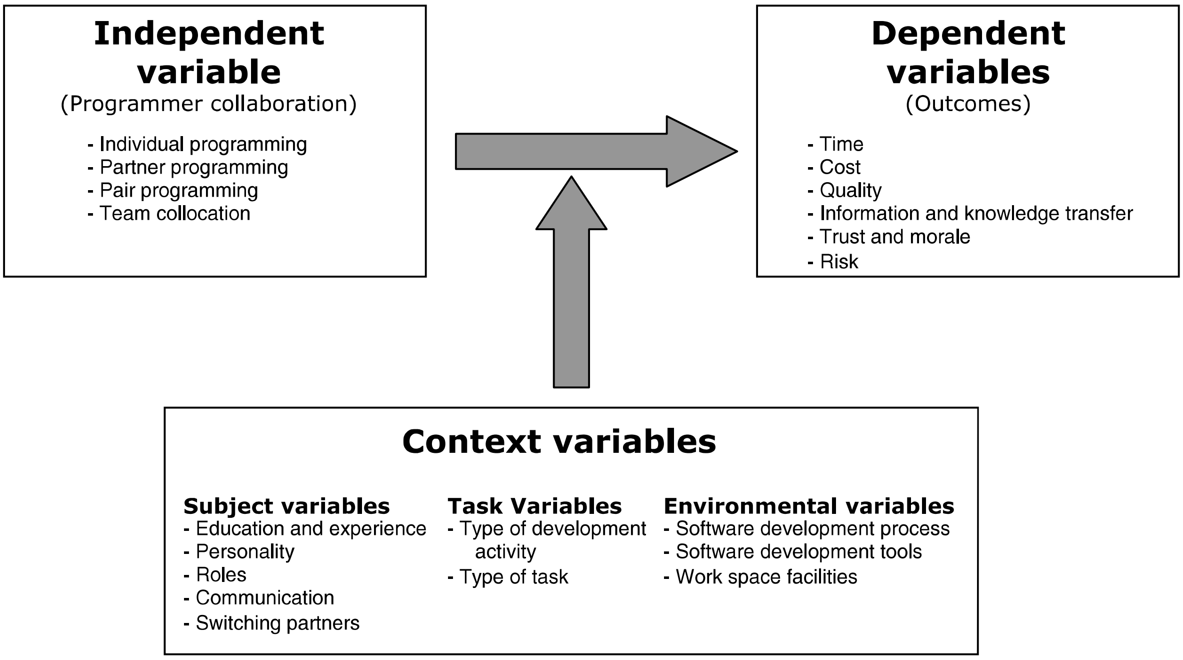


Figure 19 – Framework for research on pair programming of Gallis, Arisholm and Dyba (2003)

3.4.3 Framework on eXtreme Programming practices

Williams *et al.* (2004) defined an **evaluation framework** for eXtremme Programming (XP) practices. Their framework consists of the three models in Table 17: *context factors*, *practice adherence metrics* and *outcome measures*. Basically, these models list metyrics and project characteristics related to XP practices. The framework is focused on industrial case study research, i.e. practitioners using the framework to collect data from ongoing projects or to annotate projects that have been completed.

Similarly, Morrison (2015) proposes the establishment of an evaluation framework for software development security practices. Like the framework on XP practices, his framework should also be composed by project context factors, practice adherence metrics and outcome measures.

3.5 Final remarks

This chapter provided an overview about experimentation, according to the specificities of the Software Engineering area. We discussed some methodology-related concepts, such as the experimental process (JURISTO; MORENO, 2001; WOHLIN *et al.*, 2012) and domain-specific experimental frameworks (BASILI; SHULL; LANUBILE, 1999; GALLIS; ARISHOLM; DYBA, 2003; WILLIAMS *et al.*, 2004; WILLIAMS; LAYMAN; ABRAHAMSSON, 2005; MORRISON, 2015).

Table 17 – XP Evaluation Framework (WILLIAMS *et al.*, 2004)

Context factors	**Adherence metrics**
Sociological factors	*Planning adherence metrics*
Team size	Release length
Team education level	Iteration length
Experience level of team	Requirements added or removed
Domain expertise	Stand up meetings
Language expertise	Short releases
Experience Proj Mgr	Onsite customer
Specialist Available	Planning game
Personnel Turnover	
Morale factors	*Testing adherence metrics*
	Test coverage
Project-specific factors	Test run frequency
New & changed user stories	Test class to story ratio
Domain	Test LOC / source LOC
Staff months	Test-first design
Elapsed months	Automated unit tests
Nature of project	Custom acc tests
Constraints	
New & changed classes	*Coding adherence metrics*
Total classes	Pairing frequency
New & changed methods	Inspection frequency
Total methods	Solo frequency
New or changed KLOEC	Pair programming
Component KLOEC	Refactoring
System KLOEC	Simple design
	Collective ownership
Ergonomic factors	Continuous integration
Physical layout	Coding standards
Distraction level of office space	Sustainable pace
Customer communication	Metaphor
Technology factors	**Outcome measures**
Software development methodology	
Project management	Response to customer change
Defect prevention & removal practices	Internally-visible quality
Language	Externally-visible quality
Reusable materials	Productivity
	User stories / staff-month
Geographic factors	KLOEC / staff-month
Customer cardinality and location	Putnam product parameter
Supplier cardinality and location	Customer Satisfaction
	Morale (via survey)

The experimental framework is the basic structure of an experiment in a given domain. The framework of Basili, Shull and Lanubile (1999) consists of an instantiable goal and models of the values that can be used to instantiate it. The framework of Gallis, Arisholm and Dyba (2003) presents the structure of variables to be selected in an experiment. Finally, Williams *et al.* (2004) present a framework composed by context factors, adherence metrics and outcome measures.

Although the three frameworks seem to present different structures, they are all modeling the same kind of information about each domain. First, they all model input variables of the domain (which could be selected as independent variables/factors in a given experiment): the *process* model (Figure 16) in the reading techniques framework (BASILI; SHULL; LANUBILE, 1999), the model of *independent variable* in the pair programming framework (GALLIS; ARISHOLM; DYBA, 2003) and *adherence metrics* (Table 17) in the XP framework (WILLIAMS *et al.*, 2004). In the same way, it possible to note that the three frameworks model context factors and outcome variables from each domain. Therefore, this structure of variables is the chosen "format" for the experimental framework established in this PhD thesis.

CHAPTER 4

SURVEY ON TESTING EDUCATION

The domain we chose to explore in ths PhD thesis is related both to programming and testing education. Hence, we conducted a study to explore testing education, aiming to investigate what testing topics need to be reinforced in computing curricula from different institutions, specially by means of practical programming activities.

Software testing is among the computing areas in which graduates present more knowledge deficiencies, especially when considering industry needs (CARVER; KRAFT, 2011; RADERMACHER; WALIA, 2013). According to Radermacher and Walia (2013), previous studies in the literature do not provide specific information about which testing topics raise knowledge deficiencies. In this scenario, we investigated in details what are the curriculum-based knowledge gaps in software testing, by surveying graduates from computing programs in Brazil (SCATALON *et al.*, 2018).

This chapter presents how the survey was conducted and the obtained results. In Section 4.1, we discuss similar surveys with graduates/practioners about testing practices. In Section 4.2, we discuss the survey design. In Section 4.3 we point out some threats to validity raised by our choices in the survey design. Finally, results are presented and discussed in Section 4.4.

4.1 Related work

There are several studies comparing what is covered by computing education and what are software industry needs. This kind of investigation is relevant because there is a high demand for qualified software professionals. Attempting to align computing education with industry practices is a good way to address this demand.

Moreno et al. (MORENO *et al.*, 2012) conducted a comparison between curricular guidelines and job profiles. They identified the relationships between recommended computing competences and relevant skills to software professionals. Their results indicate that even curriculum guidelines do not cover all the core knowledge needed by professionals to perform their jobs in industry. This means that, when implemented in specific colleges or universities, these curriculum guidelines would cause knowledge deficiencies in graduates.

Similarly, Radermacher and Walia (RADERMACHER; WALIA, 2013) conducted a systematic literature review looking for knowledge deficiencies reported by previous studies. A knowledge deficiency is defined by them as any knowledge or skill that industry expects an entry-level practitioner to have and she/he lacks it. The same definition applies for academia and graduate students. The authors discussed these deficiencies in the level of Computer Science areas, such as programming, design and testing. Our study investigates in detail one of the areas mentioned by their study: software testing.

In terms of method, Lethbridge (LETHBRIDGE, 2000) conducted a study which is more similar to the one described in this chapter. The author conducted a survey with software professionals from several countries to assess the importance of computing topics (data structures, software design and patterns etc) to their career. *Testing, verification, and quality assurance* was among the topics considered more important to respondents. Also, it was among the topics that are more learned in the job, as opposed to learned in formal education. Kitchenham et al. (KITCHENHAM *et al.*, 2005) performed a similar survey in the context of UK universities.

Additionally, there are several surveys about software testing practices, but focusing only in industry. They vary in scope, ranging from multiple aspects of testing practices (NG *et al.*, 2004; GAROUSI; ZHI, 2013) to a single type of test (RUNESON, 2006; ENGSTRöM; RUNESON, 2010) and target practitioners from different nationalities, like Australia in (NG *et al.*, 2004) and Canada in (GAROUSI; ZHI, 2013). In general, they were all aiming to get a snapshot of current testing practices in industry.

Our survey explored testing practices adopted by practitioners in industry, but we also explored the software testing education delivered to them in undergraduate courses. The idea is to investigate whether the topics addressed in software testing education have been applied by the respondents in their jobs. It was directed to Brazilian practitioners and the questionnaire covered multiple aspects of software testing.

4.2 Survey design

We followed the guidelines of Kitchenham and Pfleeger to design and execute our survey (KITCHENHAM; PFLEEGER, 2008). The goal was to investigate the following research question:

What are the knowledge gaps in testing topics faced by graduates with respect to industry needs?

In order to answer it, we needed data from two different contexts: software testing education and testing practices in industry. Therefore, our strategy was to collect data about these two contexts from practitioners that are graduates from computing undergraduate programs.

We recruited respondents by sending emails with a link to a web-based questionnaire. We took advantage of mailing lists for graduates from several Brazilian universities and also the Brazilian Computer Society mailing list.

The questionnaire was composed by two sections (see Appendix A). In the first section we were seeking to find out about respondents' educational and professional background. Regarding education, we collected respondents' major and asked which computing courses they took that addressed software testing. About the professional profile, we collected their current position in the company, years of experience in software development and the programming languages generally used in their projects.

The second section had the purpose to evaluate the knowledge gaps in software testing. To this end, we collected data about respondents' undergraduate education and industry practice in testing. Then, we compared their responses for these two contexts to obtain the gaps. Still, we considered two kinds of knowledge gaps: in concepts ($\mathbf{gap}_C$) and in practice activities ($\mathbf{gap}_P$). Therefore, questions from the second section presented software testing topics to respondents and asked them to check the following options for each one:

- Industry: if they have applied the testing topic in their job.
- Concepts in education (Education_C): if during their major they have learned about the theory of the testing topic.
- Practice activities in education (Education_P): if during their major they have completed hands-on activities (such as programming/testing assignments) that gave them the opportunity to put the testing topic into practice.

We took the topics from the textbook on software testing of Delamaro et al. (DELAMARO; MALDONADO; JINO, 2016), which provides a set of testing topics that are usually addressed in computing courses. We also considered the questionnaire used in Garousi and Zhi's survey (GAROUSI; ZHI, 2013), which helped to organize the topics by characteristics of the testing activity (types of systems under test, testing levels, test types, testing approach in the development process and test case generation techniques).

In particular, we included a question asking respondents to mention which testing tools/frameworks they have used in their company. Additionally, at the end of the questionnaire, there was an optional question where respondents could add comments about their experience with software testing education and testing practices in industry.

Regarding how we calculated the knowledge gap for each testing topic, firstly we assigned values for the options Industry, Education$_C$ and Education$_P$. When a respondent had checked the option, the assigned value was one, and zero otherwise. In this way, following the same definition of knowledge gap from Lethbridge et al. (LETHBRIDGE, 2000), we were able to define equations to calculate the knowledge gaps for concepts and practice activities in education, respectively:

gap$_C$ = Education$_C$ - Industry
gap$_P$ = Education$_P$ - Industry

By applying these equations, we got both kinds of knowledge gaps (in concepts and practice activities) for each respondent in each testing topic. Considering the results individually like this, the possible resulting values are the following:

- gap = 0, when there is no knowledge gap. Either the testing topic was addressed in education and used in industry, or it was not addressed nor used.
- gap = -1, when there is a knowledge gap, which is a knowledge deficiency that a graduate faced while doing her/his job. She/he had to apply the testing topic at industry, but has not learned (or practiced) it during the major.
- gap = 1, when there is also a knowledge gap, but it can be considered as a "knowledge abundance", since it was addressed in education, but it was not applied in industry by the graduate.

Under the same reasoning, the overall knowledge gap for a given testing topic t was calculated by the average of gaps in that topic for all respondents:

$$gap_t = \frac{\sum_{s=1}^{N} gap_t si}{N}$$

where $gap_t si$ is the knowledge gap in topic t for the respondent s_i and N is the total number of respondents (90). Then, by applying this equation, we got the values of the average knowledge gaps within the interval $-1 < gap_t < 1$.

4.3 Threats to validity

The choices for survey design and conduction involve threats to validity, as it happens with any empirical study. The first one concerns generalization of results. Our sample of respondents are from Brazilian practitioners and results are limited to represent the educational and industry context from Brazil. Therefore, knowledge gaps calculated in our study are also limited to this context. Nevertheless, it can be a good indicative of points to be adjusted in testing education when seeking to meet industry needs.

Other threat concerns the accuracy of responses. Respondents had to remember about the studied/applied testing topics to answer the questionnaire. They had to inform about events that could have happened many years before. In this sense, knowledge gaps can be influenced by compromised memory, since they may have forgotten about details from undergraduate courses.

Also, the nature of the Software Engineering area itself can have an influence on results, since it presents a quickly changing landscape. In this way, a knowledge gap could simply indicate an outdated technology. This situation can apply specially for recent graduates.

Lastly, we considered knowledge gaps from the industry viewpoint. However, meeting industry expectations is not the only purpose of computing education, which should provide a good theoretical foundation that will always be used indirectly by graduates to learn about new technologies. Furthermore, computing education should develop other kinds of students' abilities besides the technical ones, such as communication, teamwork and ethics (RADERMACHER; WALIA, 2013).

4.4 Results

In this section we present the survey results. We received 90 responses from graduates in total.

4.4.1 Educational profile

Aiming to get an overview of the educational context from where we are assessing the knowledge gaps, respondents were asked to provide their academic major (see Figure 20). Computer Science is the major from over half (63.33%) of the respondents, followed by Information Systems (16.67%) and Computer Engineering (8.89%). Some respondents (11.11%) mentioned other majors, such as Data Processing, Electrical Engineering, and Software Analysis and Development.

Next, in order to understand specifically the context of software testing education, they were asked to inform which courses they took that addressed software testing (see Figure 21). Different computing courses can address this subject, so this was a question allowing multiple answers.

One interesting way to analyze these results is by noticing how much of software testing is addressed in entry-level and upper-level courses. Only 20% (18) of the respondents learned about software testing early in the curriculum, during introductory programming courses. In contrast, 82% (74) learned about testing in the Software Engineering course. Moreover, for 50% (45) of the respondents it was the only course that addressed software testing.

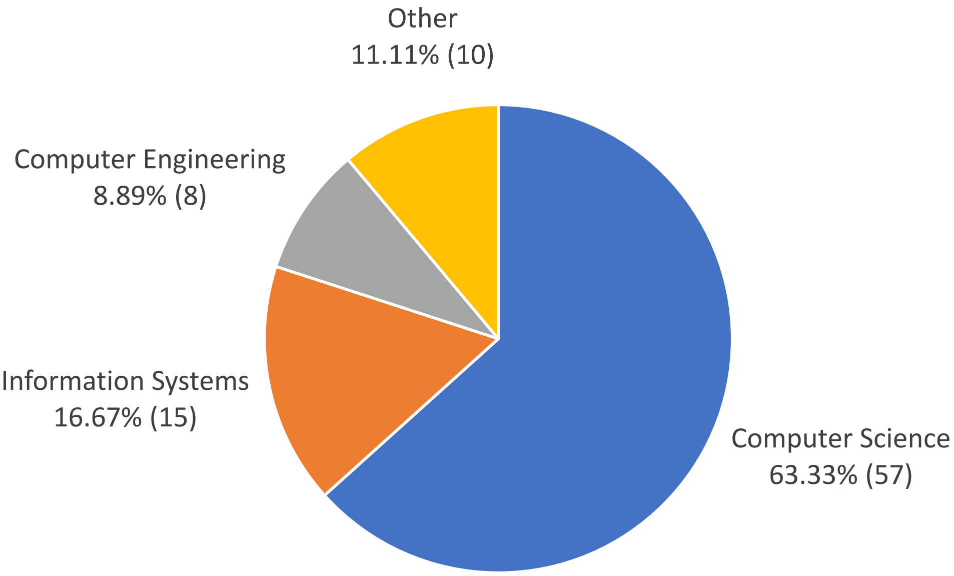


Figure 20 – Respondents' major

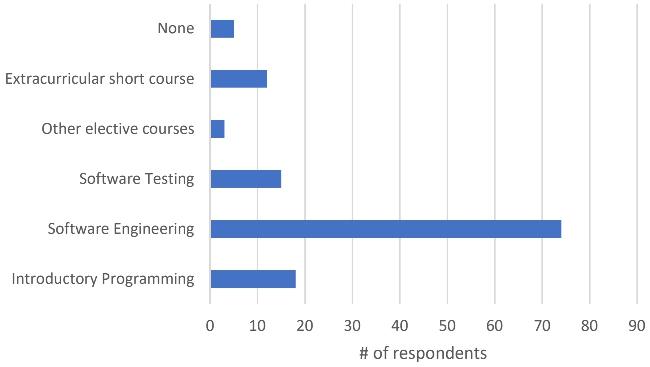


Figure 21 – Courses that addressed Software Testing in respondents major

Fewer respondents (17% – 15) took a course dedicated to this subject or other elective courses (3% – 3). Some of them (13% – 12) took short courses not included in the curriculum that addressed testing. For 6% (5) of the respondents this subject was not even addressed during the major.

4.4.2 Professional profile

Figure 22 shows respondents' current positions. Many of them are software developers (40 respondents). A smaller group work specifically with software quality assurance (QA), as testers (15) or as QA analyst/lead (14). Some of the respondents work in other roles, such as project manager (9), product owner (8) and scrum master (2).

The distribution of work experience in years is given in Figure 23. Most of them (82% – 72) had up to ten years of work experience. The average was 7.32 years, with a standard deviation of 5.91 years and a median of 7 years. There was a significant variability in respondents'

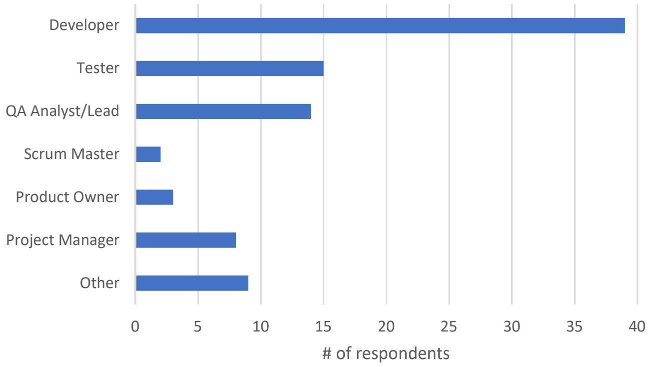


Figure 22 – Respondents' current position in industry

experience and this can contribute positively to the study, since professionals in different moments of their career can bring complementary contributions to the results.

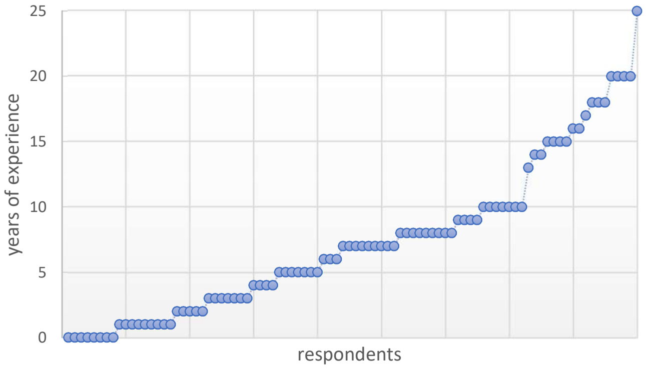


Figure 23 – Respondents' years of experience in industry

We also collected the programming languages used in the projects that respondents are involved, which are showed in Figure 24. Most respondents (71% – 64) mentioned working with Java, followed by other languages like JavaScript, Python and C++. The choice of programming language is important to the software testing activity, since each one has different kinds of existing supporting tools.

4.4.3 Knowledge gaps on software testing

Table 18 presents the knowledge gaps for testing topics, considering the average among all respondents. Similarly to individual gaps, when the value is negative, it means there is a knowledge deficiency in that topic, either in terms of concepts ($\mathbf{gap}_C$) or practice activities ($\mathbf{gap}_P$). When the value is positive (highlighted in bold), it means there is a knowledge abundance in that topic.

It is possible to note that all gaps related to practice activities ($\mathbf{gap}_P$) are negative, indicating that there is a lack of practice in software testing education. The values of knowledge

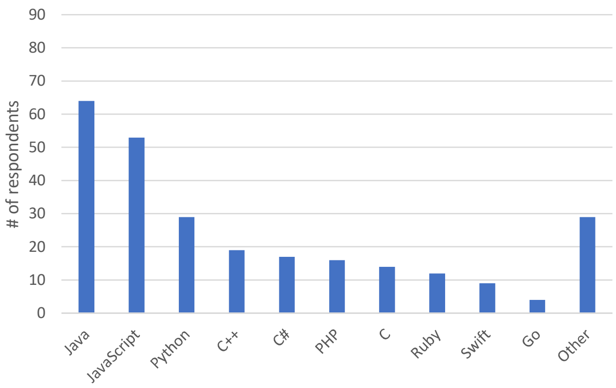


Figure 24 – Programming languages used in respondents' projects

gaps in concepts (**gap**$_C$) allow to assess if the coverage of testing topics is adequate. A negative value suggests that the corresponding testing topic is being underemphasized. Similarly, a positive value indicates an overemphasized testing topic.

Most testing topics present knowledge deficiencies (negative gaps/values). But focusing on the higher absolute values, we highlight a significant knowledge deficiency in the topic of Web applications (for both concepts and practice), for practice in all testing levels (unit, integration, system and regression testing) and practice of using client requirements/user stories as a test case generation technique.

On the other hand, knowledge abundance occurs only for concepts in the following testing topics: test of aspect oriented software and most test case generation techniques (cause-effect graph, finite state machine, control flow graph, data flow analysis and mutation analysis). Therefore, the results suggest that these particular topics have not been used much in practice, at least in the respondents' companies.

Additionally, the positive knowledge gaps in test techniques may be due to the fact that some of them work better with a mature set of requirements, which is often not true in software development. In the same direction, it is possible to note a higher demand for functional testing (category partitioning and boundary value analysis) and writing test cases from requirements/user stories.

It is interesting to point out that many respondents indicated behavior-driven development (BDD) as an approach to undertake testing during the development process (NORTH, 2006). We do not have the knowledge gap for this particular testing topic, since it was not included in the textbook contents.

Even so, it is probably a topic that should be more addressed in software testing education, specially considering its relation with other topics that presented significant negative gaps (functionality testing and the use of client requirements/user stories to generate test cases). Many BDD-related tools were also mentioned by respondents (Section **??**).

Table 18 – Knowledge gaps on software testing

Testing topic	**gap**$_C$	**gap**$_P$
Types of systems under test		
Web applications	-0.53	-0.67
Mobile applications	-0.36	-0.46
Object oriented software	-0.16	-0.50
Aspect oriented software	**0.07**	-0.08
Concurrent programs	-0.02	-0.20
Testing levels		
Unit testing	-0.18	-0.56
Integration testing	-0.44	-0.82
System testing	-0.40	-0.71
Regression testing	-0.40	-0.66
Test types		
Functionality testing	-0.37	-0.69
Performance testing	-0.48	-0.69
GUI testing	-0.30	-0.51
Usability testing	-0.09	-0.36
Security testing	-0.16	-0.41
User acceptance testing	-0.24	-0.51
Testing approach in the development process		
Test-driven (first) development (TDD)	-0.19	-0.42
Test-last development	-0.26	-0.44
Test case generation techniques		
Client requirements/user stories	-0.27	-0.58
Category partitioning	-0.07	-0.32
Boundary value analysis	-0.13	-0.37
Cause-effect graph	**0.17**	-0.13
Finite state machine	**0.29**	-0.02
Control flow graph	**0.19**	-0.04
Data flow analysis	**0.18**	-0.18
Mutation analysis	**0.20**	-0.06

4.4.4 Supporting tools

Since there was a high number of different tools (83 in total), results are given in Figure 25 sorted by categories. There was a prominence of Web application testing tools (such as Selenium and JMeter) and XUnit frameworks (such as JUnit and unittest), mentioned by, respectively, 47.8% (43) and 42.2% (38) of respondents.

4.4.5 Respondents' experiences

The last survey question took free-text answers about respondents' experiences with testing practices in industry and the software testing education delivered to them during the

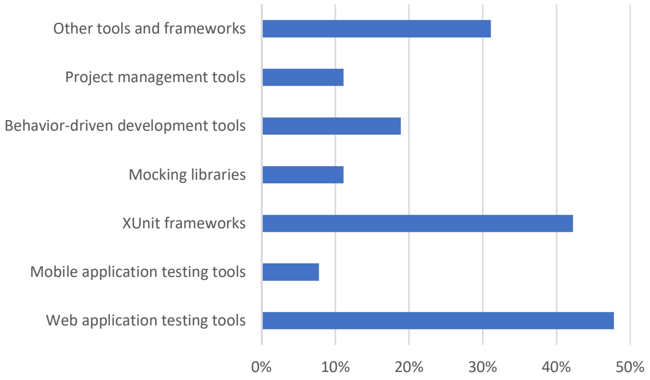


Figure 25 – Tools used in respondents' projects

major. There was a response rate of 27% (24) in this question. We identified four main points in their responses:

1. **Lack of testing practice activities in computing courses**. Many practitioners reported learning in undergraduate courses about the importance of why we should test software and the basics of testing concepts. They recognized that a sound knowledge in testing fundamentals indeed help in becoming a good tester.

   However, they complained about software testing education being too much theoretical, with a lack of practical scenarios to show students how the concepts should be applied and how software testing would have an impact in the medium and long term.

   As a result, students graduate with little actual testing skills, or depending on how the curriculum is designed, none whatsoever. Some responses are given next, in a free translation to English:

   "During my major, the subject of software testing was not addressed in a detailed way. It was restricted to only high-level concepts in the software engineering course. Almost all the learning I had about testing took place in industry"

   "I faced difficulties in adapting myself to industry needs, since it usually required experience with BDD or TDD, but I had never done any testing in practice"

   "When I started working as a QA analyst, I remember I had the feeling: 'I never saw any of this during my major, except learning the existence of these types of tests' "

   "It would be very useful if students were encouraged to submit their assignments with tests, at least in upper-level courses"

2. **Distance between software testing education and real-world practices**. Some respondents reported a good coverage of testing concepts in the major and some did not. This variation may be due to differences on how computing curricula are designed in different universities. But, similarly to what was advocated by Lethbridge et al. (LETHBRIDGE

et al., 2007), there was an agreement that testing education is distant from real-world practices:

"Unfortunately the testing activities explored in academia still are distant to most of what is applicable in industry"

"Undergraduate courses gave me a good notion of industry practices, but they evolved quickly"

"I seldom applied in my job what I learned from my major, because it is distant of industry reality in general"

"The concepts I learned during my major were good, but I think there was a lack of applying them in real situations."

3. **Software testing culture in industry**. It is interesting to notice the variation in the software testing culture from each company. While some respondents reported learning about testing in industry, thereby implying a good testing culture in their company, some of them explicitly tell about a poor testing culture:

"Many companies and development teams do not value at all the testing activity. Only technically strong teams seem to encourage it."

"In my experience, software testing was more present in the major than in the company where I work, which only cares about system testing."

4. **Factors that lead to design ineffective test suites**. Respondents pointed out how other development phases, such as analysis and design, are crucial to software testing. Additionally, they mentioned the importance to develop a tester mindset, which is only possible through practice:

"Testing requires a lot of practice in order to be really effective and not highly coupled with the application design. There is also a high deficiency in relation to software design, which should ease the testing in the first place"

"Requirements analysis is critical to create effective test cases and this depends on the experience of the test analyst. Test techniques help, but without a mature set of requirements, even with a good application of testing techniques and criteria, the constructed tests will be poor"

"I took a course that addressed well unit testing. But regarding functional testing and what concerns developing a tester mindset, it was a very superficial approach."

4.5 Final remarks

In this chapter we presented an investigation about graduates' knowledge gaps in software testing, considering industry needs. We conducted a survey with software professionals, collecting data about the testing education delivered to them and about testing practices they have applied in industry.

We considered knowledge gaps in two aspects: in concepts and in practice activities. This distinction allowed us to assess knowledge gaps in terms of how the teaching of theory and practice in software testing has been addressed in computing undergraduate programs. Additionally, the knowledge gaps were represented by values that range from -1 to 1. This raises two kinds of knowledge gaps: knowledge deficiency (negative gap) and knowledge abundance (positive gap). They can indicate, respectively, when topics are being underemphasized and overemphasized.

Ideally, all gaps should be close to zero in order to reflect a good software testing education according to industry needs. However, there are clear limitations about time and resources that do not allow to address every topic of a given subject in depth. Even so, considering that results show positive and negative gaps, there is room to counterbalance them.

In general, results indicated a deficiency for all testing topics in practice activities. In particular, there were also negative gaps in topics such as test of web applications, functionality testing and test case generation from client requirements/user stories. On the other hand, there were some positive gaps on topics like test on aspect oriented software and some test case generation techniques (cause-effect graph, finite state machine, control flow graph, data flow analysis and mutation analysis). Therefore, these topics could be considered in order to make adjustments in software testing education, when seeking to reduce graduates' knowledge gaps.

We also collected comments on respondents' experience with software testing education and industry testing practices. In summary, from the educational point of view, they reported a lack of testing practice activities in computing courses (as results about knowledge gaps have also suggested) and a distance of testing education from real-world practices. This distance was also reported in the broader context of Software Engineering by Lethbridge et al. (LETHBRIDGE *et al.*, 2007).

From the industry point of view, they reported about testing culture in the companies, which not always encourage practitioners to test software, and they pointed out some factors that can lead to ineffective test suites, such as changing requirements, bad software design and the lack of a tester mindset. Therefore, these factors should also be considered when trying to improve software testing education.

In particular, the strategy to address software testing earlier in the curriculum might help to deal with the negative knowledge gaps, which were present for all testing topics. Introductory courses provide an adequate context to encourage the use of testing practices, since students are constantly working on programming assignments (BARBOSA *et al.*, 2008; WHALLEY; PHILPOTT, 2011).

CHAPTER

5

SOFTWARE TESTING IN PROGRAMMING COURSES: A SYSTEMATIC MAPPING

Because industrial software practitioners interact with testing, whether or not they hold a 'testing' job, they need to acquire testing skills (GAROUSI; ZHI, 2013). Even so, we are graduating students from computing programs who have deficiencies in software testing skills (CARVER; KRAFT, 2011; RADERMACHER; WALIA, 2013).

A way to deal with this issue is to address software testing earlier in the computing curriculum, beginning in introductory programming courses (EDWARDS, 2003b). The idea is to provide students the opportunity to develop their testing skills incrementally throughout the curriculum (JONES, 2001). Moreover, knowledge of testing can help students improve their programming skills (EDWARDS, 2004; JANZEN; SAIEDIAN, 2008; WHALLEY; PHILPOTT, 2011).

However, the integration of software testing in this context is not straightforward, since there are many different ways to design the introductory programming sequence, as discussed in Chapter 2. Therefore, we conducted a systematic mapping of the literature to investigate the integration of testing into this diverse context and provide an overview of the research performed in the area (SCATALON *et al.*, 2019). Moreover, the mapping study is the means to leverage the different structures of existing studies in this domain in order to establish the proposed experimental framework.

This chapter describes the conducted systematic mapping and the obtained results. Section 5.1 describes our research questions and the protocol we followed to conduct the systematic mapping. Section 5.2 presents the selected studies and provides answers to the proposed research questions. Section 5.3 discusses the obtained results and, finally, Section **??** presents conclusions, threats to validity and provides directions to future work.

5.1 Research method

We followed the guidelines of Petersen et al. (PETERSEN *et al.*, 2008) to define the research protocol and to conduct the study. Briefly, we performed the following steps:

- definition of review scope (Section 5.1.1);
- search and selection of relevant papers (sections 5.1.2 and 5.1.3);
- definition of a classification scheme, composed by the categories for the mapping (Section 5.1.4), and
- data extraction from selected papers and mapping to the defined categories (Section 5.1.5).

5.1.1 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?

RQ3: How researchers have designed experimental studies on the integration of software testing in programming courses?

RQ3.1: What independent variables (factors) were selected?

RQ3.2: What dependent variables (results) and metrics were used?

RQ3.3: What context variables were considered? (The independent and dependent variables are related to the software testing educational approach and the context variables are related to the programming course context)

RQ4: How software testing has been integrated into introductory programming courses?

RQ4.1: How instructors have been teaching testing concepts in programming courses?

RQ4.2: How testing practices have been applied in practical assignments?

RQ4.3: Which kind of tools have been used to support the integration of software testing into this context?

5.1.2 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 (WOHLIN, 2014).

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 (ZHANG; BABAR; TELL, 2011).

We constructed the search string following the approach by Zhang et al. (ZHANG; BABAR; TELL, 2011). 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*)

5.1.3 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 (RADERMACHER; WALIA, 2013), we only selected papers since 2000, because papers published earlier than that would not represent current educational practices.

---

[1] <http://dl.acm.org>

[2] <http://ieeexplore.ieee.org>

[3] <http://sciencedirect.com>

[4] <http://scopus.com>

[5] <http://link.springer.com>

5.1.4 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. (PETERSEN *et al.*, 2008). 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. (AL-ZUBIDY *et al.*, 2016) 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).

5.1.5 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. The PhD student did the reading and extraction of these elements for the selected papers.

5.2 Results

Figure 26 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 19 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, 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).

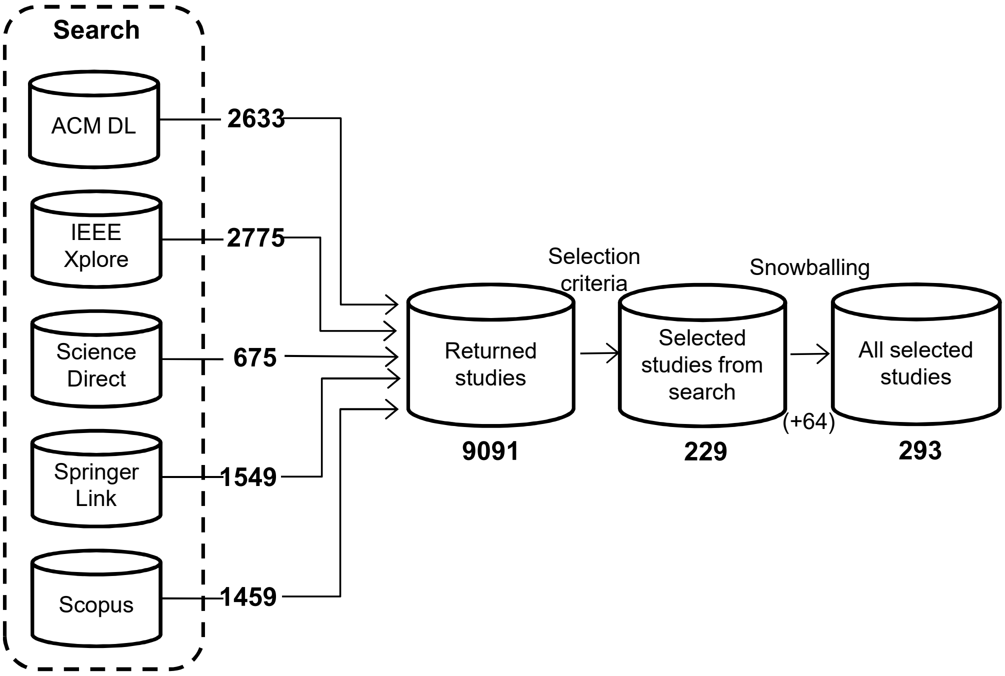


Figure 26 – Results

The following subsections provide an analysis of the identified papers. A full list of the papers along with the corresponding study number can be found in Appendix B.

Table 19 – 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

5.2.1 RQ1: Investigated topics

We identified nine *investigated topics* in the selected papers. Figure 27 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.

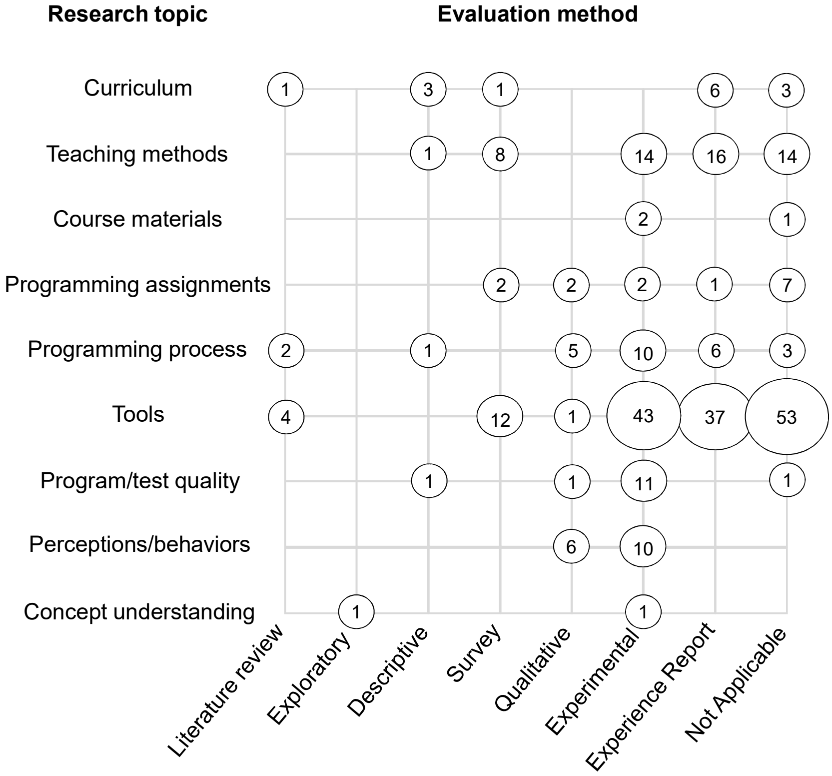


Figure 27 – Map of research on software testing in introductory programming courses

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.

5.2.1.1 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.

5.2.1.2 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.

5.2.1.3 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.

5.2.1.4 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.

5.2.1.5 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).

5.2.1.6 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).

5.2.1.7 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.

5.2.1.8 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.

5.2.1.9 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).

---

[6] <web-cat.org/questionbank>

5.2.2 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 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).

5.2.3 RQ3: Experimental design

To answer RQ3, we performed the data extraction of the papers mapped to *survey*, *qualitative* and *experimental*. All these kinds of studies involve a planned design by the researcher to collect data. So the idea is to identify the variables used in or suggested by the studies that have been conducted in this domain.

5.2.3.1 RQ3.1: Independent variables

Table 20 lists the identified independent variables and the respective levels/treatments. It is possible to notice several blank entries for the independent variables. The reason is that, for the corresponding studies, the manipulation of input variables was not clear in the paper. Many papers consist of case studies in which the authors describe the teaching method and collect data of its application in the classroom, without discussing variable selection. For some cases it was possible to identify levels/treatments from the analysis section of the paper, since results were divided into two groups, which could represent treatments.

Table 20 – Independent variables

Study	**Variable**	**Levels/treatments**
Edwards (2003a), Edwards (2004)		with/without TDD+Web-CAT
Janzen and Saiedian (2006b)		TDL / non-TDL
Janzen and Saiedian (2008)	development approach	test-first / test-last
Oliveira *et al.* (2015)		using a pascal compiler / using pascal mutants
Li and Morreale (2016)		Project A / Project B
Lemos *et al.* (2015), Lemos *et al.* (2017)	testing knowledge	with / without
Gómez, Vegas and Juristo (2016)	knowledge acquired in CS programs	1st-yr undergraduate (8%), 4th-yr undergraduate (56%), 5th-yr undergraduate (79%), 1st-yr graduate (100%)

(continued in the next page)

Independent variables (continued)

Study	Variable	Levels/treatments
Pieterse and Liebenberg (2017)	assessment method	automatic / manual
Brito *et al.* (2012)		Students who received test case sets / students who received only the program specifications
Edwards (2003d)		with/without TDD+Web-CAT
Erdogmus, Morisio and Torchiano (2005)	group affiliation	test-first / test-last
Janzen and Saiedian (2006a)		test-first programming / test-last programming
Neto *et al.* (2013)		POPT vs Blind Testing approach (non-POPT)
Camara and Silva (2016)		TDD / TDD with testing criteria
Parodi *et al.* (2016)	coding technique	Test Driven Development, Test Last, and ad hoc programming
Scatalon *et al.* (2017b)	test design task	instructor-provided test cases (IT) / student-written test cases (ST)
Gómez, Vegas and Juristo (2016)	Sw. Testing Method	black-box; white-box
Janzen and Saiedian (2007)		approach test-first (TDD) approach / test-last approach
Odekirk-Hash and Zachary (2001)		No tutor / tutor without hints / tutor with hints
Daly and Horgan (2004)		traditional method / roboprof, male / female
Thornton *et al.* (2008)		GUI vs. testing text-based assignments
Dvornik *et al.* (2011)		using WebIDE / control group using traditional static labs
Wang *et al.* (2011)		with / without AutoLEP
Buffardi and Edwards (2013b)		with / without adaptive feedback system
Janzen, Clements and Hilton (2013)		WebIDE / traditional labs
Jezek, Malohlava and Pop (2013)		old system / new system
Vujosevic-Janicic *et al.* (2013)		LAV / manual inspection
Allevato and Edwards (2014)		used Dereferee / did not use Dereferee
Buffardi and Edwards (2014b)		feedback with no hints / same number of hints / additional hints
Blaheta (2015)		CppUnit / Unci
Reynolds *et al.* (2015)		with BugFixer / without BugFixer
Braught and Midkiff (2016)		Original BlueJ / Modified BlueJ
Smith *et al.* (2017)		students trained / untrained
Buffardi and Edwards (2014a)		treatments of the adaptive feedback system (CS, DH, DS, RH, RS)

5.2.3.2 RQ3.2: Dependent variables

We also identified the variables that held experimental results in the selected empirical studies. We sorted them according to the entity being measured: **program** variables are listed in Table 21, **tests** variables in Table 22, **student/class** variables in Table 23 and **assignment** variables in Table 24.

This "classification" of entities is similar to the ones used by Juristo and Moreno (2001) (*products*, *processes* and *resources*) and by Munson (2002) (*product*, *process*, *people* and *environment*). In this way, *program* and *tests* are product entities, *student/class* is a people/resources entity and *assignment* is a *process* entity. The environment entity is equivalent to our context variables discussed in the next section.

Hence, when the identified variable was measuring a characteristic of students' programs, we classified it as a *program* variable. The same reasoning applies to students' tests and *test* variables. Still, when the variable was related to the students themselves, like measuring concept understanding, exam and quizz grades and their attitudes towards a given teaching method, it was classified as a *student/class* variable. Finally, variables related to the processes of students working on assignments and instructors assessing it are classified as *assignment* variables.

Table 21 – Dependent variables – program

Study	**Variable/metric**	**Description**
Morisio, Torchiano and Argentieri (2004)	size	LOC: total lines of code, only counts non-blank and non-comment lines inside method bodies
Morisio, Torchiano and Argentieri (2004)	size	NOC: Number of classes
Morisio, Torchiano and Argentieri (2004)	size	NOM: Number of methods
Morisio, Torchiano and Argentieri (2004)	size	AMC: Average methods per class
Morisio, Torchiano and Argentieri (2004)	size	diffLOC: Number of changed/added lines of code
Lemos *et al.* (2015)	size	
Lemos *et al.* (2017)	size	
Janzen and Saiedian (2006a)	LOC	
Cardell-Oliver *et al.* (2010)	LOC	
Denny *et al.* (2011)	LOC	
Brito *et al.* (2012)	LOC	
Buffardi and Edwards (2012a)	NCLOC	Amount of student-written solution code, in terms of the number of non-comment, non-blank lines of code
Vujosevic-Janicic *et al.* (2013)	number of lines	
Braught and Midkiff (2016)	Normalized NCLOC	The number of Non-Comment, non-blank Lines Of Code (NCLOC) in the submitted student solution normalized to the mean solution NCLOC of all first submissions to the assignment.
Janzen and Saiedian (2006a)	LOC/method	
Janzen and Saiedian (2006a)	LOC/feature	
Janzen and Saiedian (2006a)	internal quality	Nested Block Depth
Janzen and Saiedian (2006a)	internal quality	Coupling Between Objects
Janzen and Saiedian (2006a)	internal quality	Cyclomatic Complexity
Janzen and Saiedian (2006a)	internal quality	Number of Parameters
Janzen and Saiedian (2006a)	internal quality	Information Flow
Cardell-Oliver *et al.* (2010)	# classes	
Whalley and Kasto (2014)	number of operators	
Whalley and Kasto (2014)	number of unique operators	
Whalley and Kasto (2014)	number of commands	
Whalley and Kasto (2014)	average nested block depth	
Whalley and Kasto (2014)	readability metric	
Whalley and Kasto (2014)	regular expression metric	
Denny *et al.* (2011)	practiced topics	Assignment, Arithmetic, API use, Relationals, Logicals, Conditionals, Loops, Arrays

(continued in the next page)

Dependent variables – program(continued)

Study	**Variable/metric**	**Description**
Vujosevic-Janicic *et al.* (2013)	Similarity of CFGs	To evaluate structural properties of programs, we take the approach of comparing students' programs to solutions provided by the teacher
Cardell-Oliver *et al.* (2010)	# code style warnings	Code Layout (Missing whitespace, Bracket on wrong line, Line is longer than 80 characters, Construct must use brackets), Documentation (Missing a Javadoc comment, Expected Javadoc tag, Unused Javadoc tag), Java Conventions (Instance variables must be private, Java identifier must match pattern, Import warnings, More than 7 parameters), Programming Error (Conditional logic can be removed)
Denny *et al.* (2011)	cyclomatic complexity	
Whalley and Kasto (2014)	cyclomatic complexity	
Lemos *et al.* (2015)	complexity	
Lemos *et al.* (2017)	complexity	
Cardell-Oliver *et al.* (2010)	LOC/class	
Erdogmus, Morisio and Torchiano (2005)	QLTY	The quality of a story is given by the percentage of assert statements passing from the associated acceptance test suite. The quality of each story is then weighted by a proxy for the story's difficulty based on the total number of assert statements in the associated acceptance test suite. Finally, a weighted average is computed for each subject over all delivered stories, giving rise to the measure QLTY. By construction, the range of this variable is 0.5 (50 percent) to 1 (100 percent).
Brito *et al.* (2012)	program quality	a score, ranging from 0 to 10, considering the correctness of the program in relation to test case set
Edwards (2003a)	code correctness	the code correctness score measures how "correct" the student's code is. To empower students in their own testing capabilities, this score is based solely on how many of the student's own tests the submitted code can pass. No separate test data is provided by the instructor or teaching assistant. The reasoning behind this decision is that, if the student's test data is both valid (according to the instructor's reference implementation) and complete (also according to the reference), then it must do a good job of exercising the features of the student program

(continued in the next page)

Dependent variables – program(continued)

Study	Variable/metric	Description
Edwards (2003d)	code correctness	
Edwards (2004)	code correctness	
Buffardi and Edwards (2012a)	Final correctness	Correctness of solution code on student's final submission for a project, as determined by instructor-written tests
Jezek, Malohlava and Pop (2013)	correctness	
Buffardi and Edwards (2013a)	Correctness	
Souza, Isotani and Barbosa (2015)	Program correctness	Assesses whether there are defects in the student's program which are revealed by the instructor's test cases. It is equivalent to the coverage of non failured test cases for the PSt – TInst execution.
Lemos *et al.* (2015)	correctness	
Lemos *et al.* (2017)	correctness	
Scatalon *et al.* (2017b)	correctness	pass rate of student solution code
Cardell-Oliver *et al.* (2010)	tests passed (P)	
Edwards *et al.* (2012)	program performance	Fraction of test cases passed by a program in all-pairs testing
Neto *et al.* (2013)	# PT	The number of test cases (defined by the professor) that passed against the program submitted by the student
Fidge, Hogan and Lister (2013)	Code functionality	The students' classes were compiled together with our own 'ideal' unit test suite. (As explained below, this often exposed students' failures to match the specified API.) Our unit tests were then executed to determine how well the students had implemented the required functionality in their program code. The proportion of tests passed was used to calculate a 'code functionality' mark and a report was generated automatically for feedback to the students.
Utting *et al.* (2013b)	success by method	
Utting *et al.* (2013b)	# unit tests passed	
Edwards and Shams (2014a)	pass rates	pass rates of student programs when tested with the master test suite
Braught and Midkiff (2016)	% Reference Tests Passed	The percentage of instructor generated unit tests that were passed by the student's solution.
Sauvé, Neto and Cirne (2006)	compliance rate	percentage of user stories delivered with all acceptance tests passing
Utting *et al.* (2013b)	# of methods working	
Edwards (2003a)	test case failures from master suite	
Edwards (2004)	test case failures from master suite	
Wang *et al.* (2011)	Syntactic and structural defects rate	
Wang *et al.* (2011)	Functional error rate	
Morisio, Torchiano and Argentieri (2004)	defect	Defect in a program as evidences by failure of an acceptance test
Edwards and Shams (2014a)	failure rates	program failure rates for test cases in the master test suite

(continued in the next page)

Dependent variables – program(continued)

Study	**Variable/metric**	**Description**
Edwards and Shams (2014b)	failure rate	
Parodi *et al.* (2016)	Technical Debt	total number of defects identified by Findbugs / Sonar
Edwards (2003a)	defect density	To get defect density information, a selection of 18 programs were selected, 9 from each group. These programs had all comments and blank lines stripped from them. They were then debugged by hand, making the minimal changes necessary to achieve a 100% pass rate on the comprehensive test suite. The total number of lines added, changed, or removed, normalized by the program length, was then used as the defects per KSLOC measure for that program. A linear regression was performed to look for a relationship between the defects/KSLOC numbers and the raw number of test cases failed from the comprehensive test suite in this sample population. This produced a correlation significant at the 0.05 level, which was then used to estimate the defects/KSLOC for the remaining programs in the two student groups.
Edwards (2003d)	defect density	
Edwards (2004)	defect density	
Morisio, Torchiano and Argentieri (2004)	defect density	Defect/size (size as LOC)
Souza, Isotani and Barbosa (2015)	Program adequacy	Assesses whether there are unnecessary elements (i.e., statements, conditions, etc.) in the student's program. It is equivalent to the average of the covered statements and conditions for the PSt – TInst execution.

Table 22 – Dependent variables – tests

Study	**Variable/metric**	**Description**
Janzen and Saiedian (2006a)	Code Size	Test LOC
Braught and Midkiff (2016)	Normalized Test NCLOC	The NCLOC in the submitted student unit tests normalized to the mean test NCLOC of all first submissions to the assignment
Buffardi and Edwards (2012a)	Final Test NCLOC	Amount of student-written test code, in terms of the number of non-comment, non-blank lines of code
Janzen and Saiedian (2008)	# asserts	
Camara and Silva (2016)	# test cases	
Krusche and Seitz (2018)	# Test cases	
Janzen and Saiedian (2006a)	Test density	Assertions/SLOC
Shams (2013a), Shams and Edwards (2013)	test quality	test coverage
Shams (2013a), Shams and Edwards (2013)	test quality	mutation score
Shams (2013a)	test quality	all-pairs testing score
Blaheta (2015)	Test suite quality	No handin; doesn't compile; no real test; light tests; all one fn; good tests

(continued in the next page)

Dependent variables – tests (continued)

Study	**Variable/metric**	**Description**
Cardell-Oliver *et al.* (2010)	test quality	P*C (tests passed * code coverage)
Edwards (2003a), Edwards (2004), Edwards (2003d)	code coverage	branch coverage
Lee, Marepalli and Yang (2017)	Statement Coverage	
Lee, Marepalli and Yang (2017)	Branch Coverage	
Janzen and Saiedian (2006a)	Test Coverage	lines and branches
Camara and Silva (2016)	statement and branch coverage	
Cardell-Oliver *et al.* (2010)	code coverage (C)	
Braught and Midkiff (2016)	Statement Coverage	The percentage of student solution lines in the submission that were executed at least once by the student's unit tests
Aaltonen, Ihantola and Seppala (2010)	test coverage	
Edwards and Shams (2014a)	coverage scores	composite coverage scores achieved by student-written test suites
Edwards and Shams (2014a)	Code coverage measures	
Spacco and Pugh (2006)	Code coverage	
Spacco and Pugh (2006)	Unique and Redundant Coverage	Unique and redundant coverage by failing test cases
Buffardi and Edwards (2012a)	Average test coverage	Percent of statements covered by tests at time of each submission to Web-CAT, averaged for each student on each project
Buffardi and Edwards (2013a)	Coverage	
Fidge, Hogan and Lister (2013)	Test coverage	For each of our own unit tests we developed a corresponding 'broken' program which exhibited the flaw being tested for. To assess the students' unit test suite against these programs, their tests were first applied to our own 'ideal' solution program to provide a benchmark for the number of tests passed on a correct solution. Then the students' unit tests were applied to each of our broken programs. If fewer tests were passed than the benchmark our marking script interpreted this to mean that the students' unit tests had detected the bug in the program. (This process is not infallible since it can't tell which of the students' tests failed. Nevertheless, we have found over several semesters that it gives a good, broad assessment of the quality of the students' unit test suites.) The proportion of bugs found was used to calculate a 'test coverage' mark and a feedback report was generated automatically.

(continued in the next page)

Dependent variables – tests (continued)

Study	**Variable/metric**	**Description**
Edwards and Shams (2014b)	branch coverage scores	
Buffardi and Edwards (2012a)	Final coverage	Percent of statements covered by tests on student's final submission for a project
Aaltonen, Ihantola and Seppala (2010)	mutation score	
Edwards and Shams (2014a)	mutant kill ratios	
Edwards *et al.* (2012)	pass rates for test cases	pass rate for a test case in all-pairs testing
Edwards and Shams (2014a)	all-pairs score	
Edwards (2003a), Edwards (2004), Edwards (2003d)	test validity	the test validity score measures how many of the student's tests are accurate–consistent with the problem assignment. This score is measured by running those tests against a reference implementation provided by the instructor to confirm that the student's expected output is correct for each test case
Edwards (2003a), Edwards (2004), Edwards (2003d)	test completeness	the test completeness score measures how thoroughly the student's tests cover the problem. One method to assess this aspect of performance is to use the reference implementation provided by the instructor as a surrogate representation of the problem. By instrumenting this reference implementation to measure the code coverage achieved by the student tests, a score can be measured. In our initial prototype, this strategy was used and branch coverage (basis path coverage) served as the test completeness score. Other measures are also possible.
Souza, Maldonado and Barbosa (2011), Souza *et al.* (2014)	Test Coverage	PROGTEST compiles the student's program and test cases and calculates, by using JUNIT and JABUTI SERVICE , the coverage for the following combinations: (i) student's program against the student's test set (PSti – TSti); (ii) instructor's program against the student's test set (PInst – TSti); and (iii) student's program against the instructor's test set (PSti – TInst)
Souza, Isotani and Barbosa (2015)	Tests correctness	Assesses whether there are problems in the student's test cases by using the oracle program. It is equivalent to the coverage of non failured test cases for the PInst – TSt execution.
Politz, Krishnamurthi and Fisler (2014), Politz *et al.* (2016)	correctness	
Politz, Krishnamurthi and Fisler (2014), Politz *et al.* (2016)	thoroughness	
Souza, Isotani and Barbosa (2015)	Testing completeness	Assesses whether there are elements of the student's program (i.e., statements, conditions, etc.) which were not tested. It is equivalent to the average of the covered statements and conditions for the PSt – TSt execution.

(continued in the next page)

Dependent variables – tests (continued)

Study	Variable/metric	Description
Braught and Midkiff (2016)	% Student Tests Correct	The percentage of the student's submitted unit tests that were correct (i.e. passed when run with an instructor reference solution).
Edwards and Shams (2014b), Edwards and Shams (2014a)	bug revealing capability	Estimated Number of Detected Bugs
Shams and Edwards (2015)	defect-revealing capability	
Gómez, Vegas and Juristo (2016)	Revealed Faults	Number of faults revealed or not revealed by the test cases generated by students
Gómez, Vegas and Juristo (2016)	Unrevealed Faults	Number of faults revealed or not revealed by the test cases generated by students
Tang *et al.* (2016)	test case complexity score	Complexity is defined as C[t] = Q C[ti], where C[ti] is the complexity of the ith argument and C[ti] is the product of the average complexity of each nested component. Component complexity is the length for sequences and value for primitives.

Table 23 – Dependent variables – student/class

Study	Variable/metric	Description
Janzen and Saiedian (2008)	grades	
Rubin (2013)	Grades	
Daly and Horgan (2004)	grade in programming exam	
Reynolds *et al.* (2015)	Grades	
Souza, Isotani and Barbosa (2015)	grades	
Rajala *et al.* (2016)	scores	
Smith *et al.* (2017)	scores	
Spacco *et al.* (2013)	scores	
Utting *et al.* (2013b)	student scores	
Dvornik *et al.* (2011)	lab scores	
Dvornik *et al.* (2011)	midterm questions	
Jezek, Malohlava and Pop (2013)	Grades	
Janzen and Saiedian (2006b)	scores in exam and quiz	
Agarwal, Edwards and Perez-Quinones (2006)	pre post-test scores	
Odekirk-Hash and Zachary (2001)	post-test and the pretest scores	
Oliveira *et al.* (2015)	Groups' Performance	
Isomottonen and Lappalainen (2012)	students' performance	
Rubio-Sanchez *et al.* (2014)	Dropout rates	we consider both students that do not hand in any assignments for credit nor take exams (early dropouts), and the ones that having submitted at least one homework during the semester do not take the final exam (late dropouts)
Jezek, Malohlava and Pop (2013)	Course success rate	Students successfully finishing the course
Jezek, Malohlava and Pop (2013)	number of students interested in the course	
Teusner, Hille and Hagedorn (2017)	stopped out students	
Barriocanal *et al.* (2002)	satisfaction	via survey
Buffardi and Edwards (2014a)	behaviors/opinions	Test Early; Test Late; Small Increments; Large Portions; Test First; Test After

(continued in the next page)

Dependent variables – student/class (continued)

Study	**Variable/metric**	**Description**
Janzen and Saiedian (2008)	programmer opinions/perceptions	
Buffardi and Edwards (2013b)	student attitudes and perceptions of TDD	
Janzen and Saiedian (2008)	programmer perceptions	Choice; BestApproach; ThoroughTesting; Correct; Simpler; FewerDefects
Politz *et al.* (2014)	helpfulness of code and test reviews	
Janzen and Saiedian (2007)	programmer opinion	Choice; BestApproach; ThoroughTesting; Correct; Simpler; FewerDefects

Table 24 – Dependent variables – assignment

Study	**Variable/metric**	**Description**
Morisio, Torchiano and Argentieri (2004)	effort	Time spent by student to develop program for exam
Janzen and Saiedian (2006a)	Effort in minutes (Productivity)	Dev Effort, Dev Effort/LOC, Dev Effort/Feature
Thornton *et al.* (2008)	effort	we examined the programs submitted by students to count the number of statements written, and then looked at the proportion of statements devoted to test cases, relative to the entire solution submitted by each student
Daly and Horgan (2004)	SOLVETIME	gives the week that the full set of exercises was completed by each student, relative to the earliest time a set was completed. The first set, completed in week seven, was given a value of 1; the last set, completed in week 12, was scored 6. Students who did not complete were labeled 7.
Janzen and Saiedian (2008)	amount of time students reported they spent on the projects	
Buffardi and Edwards (2013a)	Time Remaining	The amount of time between when a submission was made and the assignment deadline. Negative values represent submissions made after the deadline.
Buffardi and Edwards (2013a)	Time Elapsed	The amount of time between the students' first submission for that assignment and the current submission in question.
Buffardi and Edwards (2013a)	Relative Worktime	The amount of time elapsed, expressed as a percentage of the total duration over all of the student's submissions for an assignment. Zero- and one-values represent the first and final submissions by that individual, respectively. This metric disregards the relationship between submission time and the assignment deadline. Instead, it represents the progression of time within the workflow of development.

(continued in the next page)

Dependent variables – assignment (continued)

Study	Variable/metric	Description
Neto *et al.* (2013)	Time	Time spent between (i) the student receiving an ill defined specification and (ii) the student submitting a program to the Bling testing system
Spacco *et al.* (2013)	estimated hours writing code	
Oliveira *et al.* (2015)	time	each student had specified a starting time and an ending time for their experiment
Li and Morreale (2016)	Planning time; Coding time; Testing time; Revision time	
Rajala *et al.* (2016)	time	
Lemos *et al.* (2015), Lemos *et al.* (2017)	time	
Matthies, Treffer and Uflacker (2017)	Time-Per-Task Analysis	An indicator of both the general difficulty of a task as well as how much effort was required by participants is the time taken to solve tasks
Morisio, Torchiano and Argentieri (2004)	productivity	Size/effort (size as LOC)
Erdogmus, Morisio and Torchiano (2005)	PROD	Productivity is defined as output per unit effort. The number of stories is well-suited for measuring output: For our purpose, it is a superior measure of real output than program size (e.g., lines of code) in that it constitutes a more direct proxy for the amount of functionality delivered. It is still an objective measure since we can compute it automatically based on black-box acceptance tests. If a story passed at least 50 percent of the assert statements from the associated acceptance test suite, then the story was considered to be delivered. The number of stories delivered was normalized by total programming effort to obtain the productivity measure PROD
Janzen and Saiedian (2008)	productivity	LOC; time
Erdogmus, Morisio and Torchiano (2005)	TESTS	The variable TESTS measures the number of programmer tests written by a subject, again, per unit of programming effort. A programmer test refers to a single JUnit test method. Through visual inspection of the subjects' test code, we filtered out ineffective tests, such as empty test methods, duplicated test methods, and useless test methods that passed trivially. Because subjects were free to work as many hours as they wanted, the variation in total programming effort was large (ranging from a few hours to as many as 25 hours). Hence, it was necessary to normalize the number of tests by the total effort expended.

(continued in the next page)

Dependent variables – assignment (continued)

Study	Variable/metric	Description
Janzen and Saiedian (2008)	#asserts/LOC	
Janzen and Saiedian (2008)	#asserts/module	
Sauvé, Neto and Cirne (2006)	average tests per user story	
Thornton *et al.* (2008)	proportion of each student's submission that was devoted to test cases over time, from the student's very first submission until their final solution	
Buffardi and Edwards (2013a)	Test:Solution Relative NCLOC NCLOC: Non-comment lines of code, separated into lines that are part of the student's solution and lines that are part of the student's software tests.	
Braught and Midkiff (2016)	TSSS (Test Statement Per Solution Statement)	The ratio of Test NCLOC to Solution NCLOC
Buffardi and Edwards (2012b)	adherence to TDD	Test Statements per Solution Statement (TSSS): the number of programming statements in student-written test classes relative to the number of statements in their solution classes, Test Methods per Solution Method (TMSM): the number of student-written test methods relative to the number of methods in their solution
Buffardi and Edwards (2012a)	student behavior and affect with regards to adhering to TDD	Test Statements per Solution Statement (TSSS), Test Methods per Solution Method (TMSM)
Buffardi and Edwards (2013b)	adherence to incremental unit testing	two metrics for evaluating testing quality and quantity over time: average coverage and average test-methods-per-solution-method (TMSM)
Thornton *et al.* (2008)	automated grading results	student programs were executed against a set of instructor-provided reference tests
Pieterse and Liebenberg (2017)	marks	Mark distribution per assessment method
Edwards (2003a), Edwards (2004)	recorded grades	"Recorded grades" represents the average final assignment score recorded in the instructor's grade book. Half of each score came from the automated assessment and half from an independent review of the student's source code by a graduate teaching assistant.
Edwards (2003a), Edwards (2004)	TA assessment	"TA assessment" reflects the average amount of credit received for the TA portion of the student's grade
Edwards (2003a), Edwards (2004)	Curator assessment	
Edwards (2003a), Edwards (2004), Edwards (2003d)	Web-CAT SCORE	To combine these three measures into one score, a simple formula is used. All three measures are taken on a 0%-100% scale, and the three components are simply multiplied together. As a result, the score in each dimension becomes a "cap" for the overall score–it is not possible for a student to do poorly in one dimension but do well overall. Also, the effect of the multiplication is that a student cannot accept so-so scores across the board. Instead, near-perfect performance in at least two dimensions should become the expected norm for students.

(continued in the next page)

Dependent variables – assignment (continued)

Study	**Variable/metric**	**Description**
Janzen and Saiedian (2008)	project evaluations	
Daly and Horgan (2004)	ROBOSCORE	ROBOSCORE gives the overall score achieved in the RoboProf exercises. It is expressed as a percentage.
Souza, Maldonado and Barbosa (2011), Souza *et al.* (2014)	Suggested Grade	
Wang *et al.* (2011)	grade	
Edwards (2003a)	# on time/late submissions	
Edwards (2003a)	time of first submission	(hours before due)
Edwards (2003a)	time from first submission until assignment due	
Politz, Krishnamurthi and Fisler (2014)	Event Time (as Hours Before Due Date)	Event Type: submit tests, receive review, read review
Thornton *et al.* (2008)	time	spent on assignment difference between the times of a student's first and last submission
Spacco *et al.* (2013)	days before deadline of first programming snapshot	
Neto *et al.* (2013)	# SV	Number of submitted versions per student
Sridhara *et al.* (2016)	Code Snapshots	
Souza, Kolling and Barbosa (2017)	# samples	
Spacco *et al.* (2013)	# snapshots	
Spacco *et al.* (2013)	# compilable snapshots	
Spacco *et al.* (2013)	Number of snapshots distributed over hours of the day	
Sridhara *et al.* (2016)	Snapshots with Fuzz Test Errors	
Nishimura, Kawasaki and Tominaga (2011)	# submissions	
Denny *et al.* (2011)	number of submissions	
Souza, Isotani and Barbosa (2015)	# submissions	
Rajala *et al.* (2016)	# submissions	
Matthies, Treffer and Uflacker (2017)	# of completed tasks	
Krusche and Seitz (2018)	# Submitting students	
Isomottonen and Lappalainen (2012)	# students who have returned weekly exercises	
Spacco *et al.* (2013)	# projects	
Spacco *et al.* (2013)	# students	
Spacco *et al.* (2013)	# students with a submission	
Sridhara *et al.* (2016)	Students Completing Project	
Daly and Horgan (2004)	AVATTEMPT	AVATTEMPT measures the average number of repeated attempts per exercise for each student; students can correct and submit programs as often as they wish. After resubmission RoboProf updates the score.
Krusche and Seitz (2018)	Submissions per student	
Jezek, Malohlava and Pop (2013)	# submits needed to successfully finish the assignment	
Krusche and Seitz (2018)	# Overall submissions	
Sridhara *et al.* (2016)	Students Attempting Project	
Sridhara *et al.* (2016)	Students Attempting Target Question	

(continued in the next page)

Dependent variables – assignment (continued)

Study	Variable/metric	Description
Rubio-Sanchez *et al.* (2014)	# erroneous exercises	
Denny *et al.* (2011)	number of failing submissions	
Denny *et al.* (2011)	number of non-compiling submissions	
Edwards, Shams and Estep (2014)	# submissions per student with non-termination problems	
Tang *et al.* (2016)	# incorrect programs on expert test set	
Vujosevic-Janicic *et al.* (2013)	Programs with bugs	
Allevato, Edwards and Perez-Quinones (2009)	causes of abnormal termination (test case failure)	categories: use of NULL pointer; use of uninitialized pointer, use of deleted pointer, use of out of bounds pointer, other
Allevato and Edwards (2014)	memory-related errors in students' submissions	Each category describes the misuse of a particular type of pointer, usually by dereferencing (uninitialized pointers, dangling pointers, null pointers)
Pieterse and Liebenberg (2017)	programs that don't compile	
Sridhara *et al.* (2016)	Incorrect Attempts at Target Question	
Spacco *et al.* (2005)	# not implemented	
Spacco *et al.* (2005)	# exception thrown	
Spacco *et al.* (2005)	# assertion failed	
Daly and Horgan (2004)	NPLAGIAR	NPLAGIAR measures the number of programming exercises that were plagiarized by each student.
Spacco *et al.* (2013)	# release tokens used in a work session	Students can upload their code to the server, spend a release token, and see the number of release tests passed and failed, as well as additional feedback about the first two release tests which failed. Furthermore, students only have 3 release tokens, and each token takes 24 hours to regenerate.
Spacco *et al.* (2013)	effect of release tokens on the score	positive, neutral, negative
Buffardi and Edwards (2015)	# hints earned/unearned revealed/obscured	
Barriocanal *et al.* (2002)	# students that developed test cases in their assignments	
Isomottonen and Lappalainen (2012)	# students who had written tests	
Buffardi and Edwards (2014b)	TYPE OF CHANGE (minor, moderate, major)	types of change to test code Adding new test methods constituted major changes, adding only new test NCLOC signified moderate changes, and any changes that did not require changing the number of test NCLOC were minor changes. We recorded changes (Δ) in non-comment lines of code (NCLOC) for the solution and the tests independently. Likewise, we measured the change in test coverage as well as changes in the number of assertions in the students' test code

(continued in the next page)

Dependent variables – assignment (continued)

Study	Variable/metric	Description
Buffardi and Edwards (2014a)	Δ coverage	how coverage on the final submission compares to that of the first submission
Buffardi and Edwards (2014a)	Δ test NCLOC	increases or decreases in the amount of test code from the first to last submission within an assignment
Souza, Kolling and Barbosa (2017)	# diffs	we counted and analyzed the number of diffs between the incorrect source codes and the fixed source codes of each sample. To do so, we considered the textual differences of the source codes. Each line of source code added, removed or changed was counted as one diff.
Souza, Kolling and Barbosa (2017)	# Statements Fixes	
Souza, Kolling and Barbosa (2017)	# Expressions Fixes	
Baumstark Jr. and Orsega (2016)	LOC changed across all revisions	
Baumstark Jr. and Orsega (2016)	# changed classes and methods	
Edwards and Li (2016)	progress indicators	1. Adding New Solution Method(s); 2. Removing Static Analysis Errors; 3. Reducing Cyclomatic Complexity; 4. Reducing Average Method Size; 5. Increasing Comments Density; 6. Increasing Solution Classes; 7. Increasing Correctness; 8. Adding New Test Method(s); 10. Increasing Number of Tests per Method; 11. Increasing Statement Coverage; 12. Increasing Method Coverage; 13. Increasing Conditional Coverage; 14. Increasing Assertion Density; 15. Increasing Test Classes
Odekirk-Hash and Zachary (2001)	TA help time	
Edwards, Shams and Estep (2014)	Test Case Execution Time	
Krusche and Seitz (2018)	Assessment time	
Yi *et al.* (2017)	Repair Rate	(#Fixed)/(#Programs)
Madeja and Poruban (2017)	Execution speed of tests	
Edwards, Shams and Estep (2014)	# test cases triggering infinite looping behavior in student submissions	
Edwards, Shams and Estep (2014)	# test cases completed before first non-termination (Timeout)	
Souza, Isotani and Barbosa (2015)	Debugging completeness	Assesses whether there are defects in the student's program which are revealed by the student's test cases, but they were not debugged and fixed. It is equivalent to the coverage of non failured test cases for the PSt – TSt execution.
Baumstark Jr. and Orsega (2016)	unit test distance (in revisions)	number of revisions between when a method is first created and when it first appears to have a written test.

5.2.3.3 RQ3.3: Context variables

The context variables are the ones not directly related to the software testing integration approach, but yet could have an influence on it. They are the variables of the programming education context where the empirical study takes place. Again, we classified the identified variables into different groups: **student** variables in Table 25, **assignment** variables in Table 26, **course** variables in Table 27, and **other practices** variables in Table 28.

Table 25 – Context variables – student

Study	Variable	Description
Hilton and Janzen (2012)	previous programming experience	
Rubin (2013)	students' familiarity and experience with programming in any language	
Rubin (2013)	learning style	VARK learning style questionnaire, which is a proven tool used to assess each student's learning preferences
Camara and Silva (2016)	previous experience on programming, TDD and software testing	
Parodi *et al.* (2016)	completed courses	
Dvornik *et al.* (2011)	prior programming experience	
Brito *et al.* (2012)	experience/no experience in programming	
Janzen and Saiedian (2007)	TDD exposure time	
Scatalon *et al.* (2017b)	prior testing habits	
Brito *et al.* (2012)	public/private school	
Denny *et al.* (2011)	gender	
Brito *et al.* (2012)	male/female	
Buffardi and Edwards (2013b)	student motivation	We considered the possibility these correlations demonstrated an effect of student motivation instead of TDD adherence.

Table 26 – Context variables – assignment

Study	Variable	Description
Neto *et al.* (2013)	level of complexity of the target program	We believe that the more complex the problem is the higher may be the benefits of POPT
Janzen and Saiedian (2007)	project size	
Buffardi and Edwards (2013a)	scale and complexity of assignments	assignments vary in scale and complexity between courses and semesters
Whalley and Kasto (2014)	difficulty	
Janzen and Saiedian (2007)	kind of programming project	small or semester-long etc
Teusner, Hille and Hagedorn (2017)	perceived difficulty of a task	depends on previous knowledge, supplied hints, the required time for solving and the number of failed attempts the participant made. Furthermore, the detail and accuracy of the problem description, the restrictiveness of the applied test cases and the preparation provided specifically for a given exercise also influence the perceived difficulty of a task
Matthies, Treffer and Uflacker (2017)	duration of exercise	
Spacco and Pugh (2006)	skeleton code/scaffolding	
Teusner, Hille and Hagedorn (2017)	additional help	
Teusner, Hille and Hagedorn (2017)	offered templates and hints	
Shams (2013a)	design freedom	However, we found that when students have larger design freedom in assignments, significant number of their tests examine components related to their personal design decisions.

(continued in the next page)

Context variables – assignment (continued)

Study	Variable	Description
Shams and Edwards (2013)	design freedom	This quality assessment strategy is at odds with open-ended assignments, or assignments with large amounts of design freedom. In those cases, student tests are so diverse that significant numbers cannot be applied to a common reference solution, resulting in an artificially depressed quality measure. Is this acceptable? 2013shams-a "a common specification" Achieving accurate quality measurement relies on students writing tests to "a common specification" as much as possible, instead of to their own personal design. Will this lead to over-constrained assignments that preclude students from creating their own designs?
Fidge, Hogan and Lister (2013)	API specification	
Buffardi and Edwards (2014a)	student's personal design	All student tests were run against an instructor-provided reference solution to weed out tests that were invalid or only applicable to the student's personal design
Teusner, Hille and Hagedorn (2017)	suitability of an exercise	depends on the specific (sub-)topics dealt with and on the (perceived) difficulty, composed of: the difficulty of the actual steps to solve the exercise, the prior knowledge of the participant 2017Teusner expressiveness of the exercise description
Janzen and Saiedian (2007)	programmer collaboration	
Missiroli, Russo and Ciancarini (2017)	"collaboration"	We only had pair teams, instead of both pairs and solo programmers
Fidge, Hogan and Lister (2013)	GUI	The GUI code in the two pair-programming assignments was not accompanied by unit tests and was marked manually.

Table 27 – Context variables – course

Study	Variable	Description
Missiroli, Russo and Ciancarini (2017)	institution	Three schools were involved instead of a single one
Whalley and Kasto (2014)	paradigm/curriculum model	Some of the metrics used in this study may not be generalizable to all teaching contexts or indeed to all novice programming tasks. Courses that adopt an objects first pedagogy may have writing tasks for which other object orientated metrics might be applicable such as cohesion and coupling metrics. For a back to basics, algorithm focused, java course that does not utilise micro worlds but instead uses a typical IDE metrics such as number of commands may not be relevant.

(continued in the next page)

Dependent variables – course (continued)

Study	Variable	Description
Gómez, Vegas and Juristo (2016)	Academic year	
Gómez, Vegas and Juristo (2016)	CS program	

Table 28 – Context variables – other practices

Study	Variable	Description
Isomottonen and Lappalainen (2012)	game development	
Rubin (2013)	Live Coding	
Rubin (2013)	problem-based learning	
Rubin (2013)	collaborative learning	
Rubin (2013)	active learning	
Politz, Krishnamurthi and Fisler (2014), Politz *et al.* (2016)	peer review	
Lee, Marepalli and Yang (2017)	Coding Dojo	Coding Dojo is a dynamic and collaborative activity where people can practice programming, especially techniques related to agile methods
Krusche and Seitz (2018)	version control	By using VCS and teaching its application, we achieve the same outcome [snapshots of students' programs]. Students commit multiple iterations of their solution, resulting in a commit history that can be evaluated
Baumstark Jr. and Orsega (2016)	use of version control	
Fidge, Hogan and Lister (2013)	Pair-programming Assignments	

5.2.4 RQ4: Teaching practices

In order to get an overview of how the integration of software testing has been done in programming courses, we extracted information about the teaching practices in all empirical studies, including experience reports (*survey*, *qualitative*, *experimental* and *experience reports*).

5.2.4.1 RQ4.1: Testing concepts in programming course materials

Very few papers address how testing concepts are taught in programming courses. Only 22.56% (44) of the empirical studies mention some kind of instruction in testing concepts. In general, the authors provide a brief description about the testing instruction, like suggested in the following text snippets from the papers: "students are taught testing best practices", "the lecture introduced automated unit testing", "instructor led introduction to the running and reading of test units", "the role of unit testing is introduced early" and so on. This kind of description does not allow to identify what testing concepts were presented to students. In contrast, the study of Elbaum *et al.* (2007) is a good example of paper that provides a thorough description of testing concepts.

It is interesting to notice that even fewer empirical studies mention the teaching of techniques/criteria to students: 6.15% (12). So, it is not clear how students learn to select test input/output values and design cases in the remaining studies. This issue is particularly important in approaches that students are supposed to write their own test cases.

5.2.4.2 RQ4.2: Testing practices in programming course assignments

We also investigated how testing practices have been integrated into programming assignments. In this direction, we identified how testing can be merged in each aspect of an assignment: description, steps, deliverables and grading.

In the assignment description testing can be a part of the problem specification, as acceptance tests, aiming to test students' programs at the system level and helping them to validate their solutions (MORISIO; TORCHIANO; ARGENTIERI, 2004; ERDOGMUS; MORISIO; TORCHIANO, 2005; SAUVÉ; NETO; CIRNE, 2006; SAUVE; NETO, 2008; MISSIROLI; RUSSO; CIANCARINI, 2017). Also, the tests supplied by the instructor can serve as a test harness or scaffolding, in the lower level of unit tests (ISOMOTTONEN; LAPPALAINEN, 2012; UTTING *et al.*, 2013b; PAUL, 2016).

Regarding the assignment steps, the instructor can provide guidelines to students about the programming process to adopt while working on the assignments. 20% (39) of the empirical studies mention that students were instructed to use a programming process. Figure 28 shows the distribution of adopted programming processes in such studies.

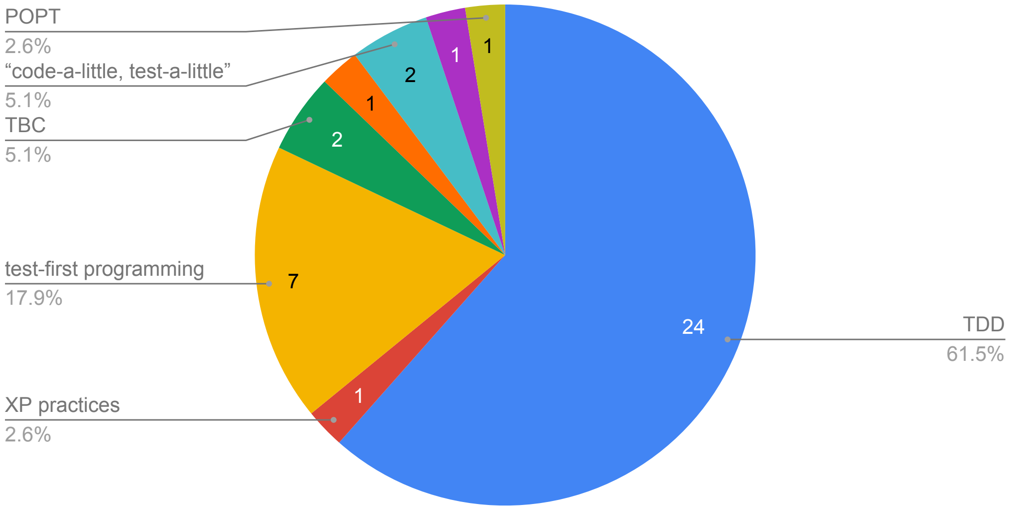


Figure 28 – Programming processes used in the empirical studies

All these processes guide students on how to bind the activities of programming and testing. In particular, the identified processes do not provide details on how the testing activity should be conducted. Because of that, we investigated how the testing activity has been conducted by students, in terms of the testing tasks listed by Ammann and Offutt (2016): test design, test automation, test execution and result evaluation.

Considering the context of students working on assignments, we marked the testing task for a given study when students were responsible for that task while completing their assignments. In this way, **test design** was marked when students were responsible for choosing input/output values and designing their own test cases, **test automation** was marked when students were supposed to code test cases, **test execution** when students executed the test cases against their programs, and **result evaluation** when students evaluated test results getting feedback about their programs. Figure 29 shows the distribution of each testing task conducted by students in the empirical studies.

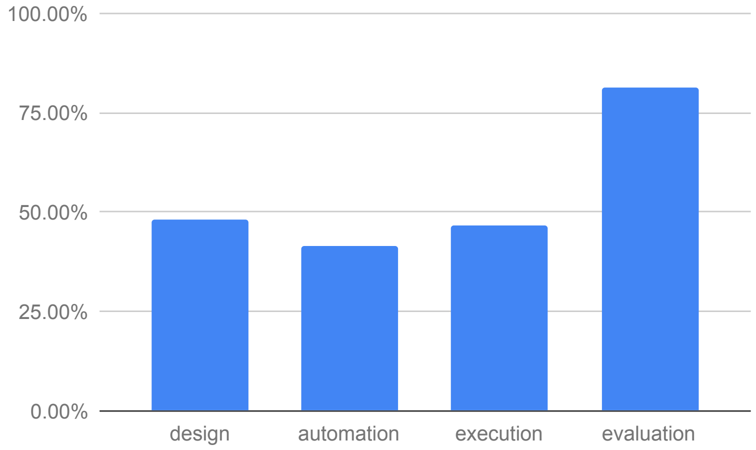


Figure 29 – Testing tasks performed by students in the empirical studies

5.2.4.3 RQ4.3: Supporting tools

74.87% (146) of the empirical studies mention the use of a supporting tool related to the integration of testing. We identified the adopted tools according to the categories we established in Section 5.2.1.6. Figure 30 shows the distribution of tools categories.It is important to note that some studies adopted more than one category of tool.

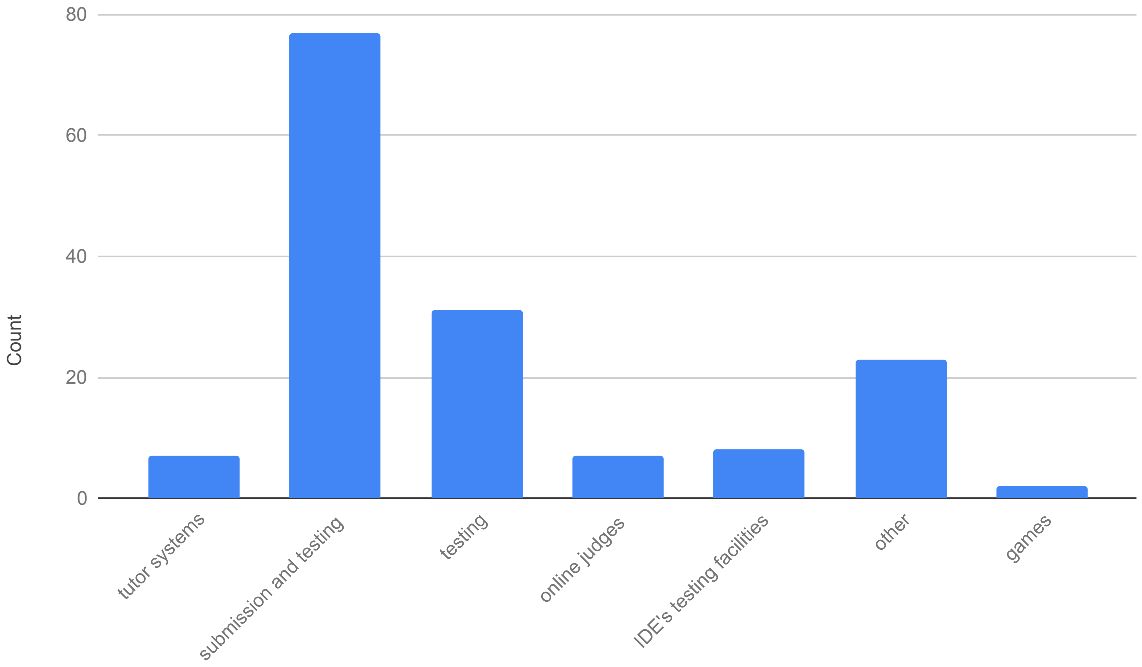


Figure 30 – Supporting tools used in the empirical studies

5.3 Discussion

The map of papers in Figure 27 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 5.2.2. 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).

5.4 Final remarks

In this chapter 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.

Still, we discussed benefits and drawbacks of the approach to the teaching of programming. We also identified teaching practices that has been used in the classroom to integrate software testing, both in terms of testing concepts taught in programming courses materials and testing practices adopted in programming assignments.

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 **??**) 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).

Moreover, we identified the variables used by researchers in empirical studies that involve planned data collection (*experimental*, *qualitative* and *survey* studies). These identified variables helped to establish the proposed experimental framework, discussed in the next chapter.

Similarly, the identified benefits and drawbacks in Section **??** motivate further research, aiming to raise the benefits and minimize the drawbacks. Also, as discussed in Section **??**, 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 (GUZDIAL, 2013).

CHAPTER

6

EXPERIMENTS ABOUT THE TEST DESIGN TASK

An analysis of students' test suites in (EDWARDS; SHAMS, 2014b) 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.

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 (FREZZA, 2002; BARBOSA *et al.*, 2003; WICK; STEVENSON; WAGNER, 2005; COLLOFELLO; VEHATHIRI, 2005; AGARWAL; EDWARDS; PEREZ-QUINONES, 2006; ELBAUM *et al.*, 2007; THURNER; BOTTCHER, 2015), where students were instructed on fundamental testing concepts.

This chapter presents the experiments conducted during this PhD work. Section 6.1 describes an experiment that compared students conducting testing with their own test cases and with instructor test cases, thus investigating the effect of students conducting the test design task on their programming performance. Section 6.2 describe other experiment we conducted investigating students' test design skills. In particular, we investigate how their testing skills progress with different modules of testing concepts completed.

6.1 Experiment on students performing the test design task

The following subsections provide details about the experiment we conducted and the obtained results, which are published in (SCATALON *et al.*, 2017a). We followed guidelines from Juristo and Moreno (JURISTO; MORENO, 2001) and Wohlin et al. (WOHLIN *et al.*, 2012) to plan and execute the study.

6.1.1 Goal

The integration of software testing in computing courses should be done in a way that adds value to the programming assignments. Students should be able to benefit from the testing practice, what can lead them to perform it more willingly.

In this scenario, we intend to investigate the integration of testing practices into programming assignments. We focused on one of the aspects of the testing activity, the test design task, and its effect on student programming performance.

If students are being able to improve their code from the feedback provided by testing results, they probably will feel the need to do so and the integration of software testing will not disrupt the normal course flow to teach the main subject, i.e. the programming concepts.

6.1.2 Subjects

We applied a subject characterization questionnaire in the beginning of the short course. In total, 21 students attended, all Computer Science majors, distributed as indicated by Table 29. 11 students provided consent on the collected data.

Table 29 – Distribution for student level (n=11)

Student level	**#**
Freshman	45.45% (5)
Sophomore	9.09% (1)
Junior	9.09% (1)
Senior	36.36% (4)

Since we focused on the application of software testing for programming assignments, we also characterized the students regarding the introductory courses they had already completed or were still attending (see Table 30).

Table 30 – Distribution for introductory programming courses (n=11)

Course	**#**
Introductory Programming I	100% (11)
Introductory Programming II	100.00% (11)
Data Structures I	54.55% (6)
Data Structures II	54.55% (6)
Object Oriented Programming	45.45% (5)

Introductory Programming I and II are first-year courses that involve the teaching of fundamental programming constructs, with an imperative-first approach using the C language.

Data Structures I and II are second-year courses about data structures (also in C) and the corresponding algorithms to operate them. Finally, Object Oriented Programming is a second-year course in which students learn about object-oriented concepts using the Java language.

Regarding their prior testing habits, according to Table 31, they usually perform testing while working on their programming assignments. However, as Table 31 shows, most students do not use or do not know what a testing criterion is. This is an interesting outcome, considering that testing practices are not addressed during regular programming courses in this setting. Also, it shows that even though students test their code, they are doing it without proper instruction in this subject.

Table 31 – Student testing habits in programming assignments

Question: *Do you test the programs you write?*	
Answer	#
I do not know what it means to test a program	0% (0)
There is no need to	0% (0)
Yes, only if there is enough time	0% (0)
Yes, I always test at least a little	90.9% (10)
Yes, I always try to test a lot before delivering the program	9.1% (1)

Table 32 – Use/knowledge of testing criteria

Question: *Do you use any testing criteria to test your programs?*	
Answer	#
Yes	45.45% (5)
No	27.27% (3)
I do not know what test criteria is	27.27% (3)

6.1.3 Experimental Objects

The experimental objects were the programming assignments from the final project proposed to students at the end of the short course. The project was composed by three assignments, which consisted in implementing alarm clock features (based on the assignments from (UTTING *et al.*, 2013a)).

In order to solve them, students had to represent time, as composed of hours and minutes (in a simplified way) and perform basic operations with time calculation.

- **Assignment 1.** Tick operation: advance the current time in one minute.
- **Assignment 2.** Alarm clock set features:
  - **(A)** Wake-up time: set the alarm with the desired wake-up time and show the user the remaining sleep time. Involves the implementation of the sum of two times.

  - **(B)** Remaining sleep time: set the alarm with the desired remaining sleep time and show user the resulting wake-up time. Involves the implementation of the subtraction of two times.

Assignment 1 was carried out as a training, with the instructor's assistance. In this way, students were able to become familiar with the testing practice and this problem context. After, they completed assignments 2A and 2B by themselves.

6.1.4 Hypotheses

We intend to compare different testing approaches during programming assignments, according to the hypotheses listed on Table 33.

In the first approach students are not responsible for the test design task, they receive ready-made instructor tests (IT). In the second one students need to perform the test design, besides the other testing tasks (test execution and evaluation). So, the test cases are written by the own students (student tests – ST).

The effect of these two testing approaches were observed in students' programming performance. In turn, programming performance was considered in terms of correctness of the program delivered by students to solve the assignment.

Table 33 – Study hypotheses

Hypotheses type	**Formalized hypotheses**
Null hypothesis	H_0: $correctness_{IT} = correctness_{ST}$
Alternative hypotheses	H_1: $correctness_{IT} > correctness_{ST}$ H_2: $correctness_{IT} < correctness_{ST}$

6.1.5 Variables

During the experiment, students were supposed to apply two different testing approaches to complete programming assignments. In order to configure these testing approaches, some aspects were kept constant for both, while the test design task was the aspect that varied.

Starting from the constant aspects, both assignments involved automated unit testing using the assert macro in C. Also, students were always responsible for the *execution* and *evaluation tasks* of the testing activity. Regarding supporting tools, students were free to use the IDE they were familiar with to write the solution code, the test cases (when applicable) and to execute test cases.

The experiment had one independent variable, the *test design task*. This variable had the two following treatments:

- *instructor-provided test cases* (IT), when students receive ready-made test cases and are not responsible for test design, but only for test execution and evaluation; and
- *student-written test cases* (ST), when students are also actively involved with the test design task and have to write their own test cases.

Since we were aiming to evaluate students' programming performance, the dependent variable was the *correctness* of the solution code submitted for the programming assignment. We measured correctness as the *pass rate* of student solution code.

This metric was calculated dividing the number of test cases for which the unit passed by the total number of test cases for that unit. We used a set of reference test cases, which was the same we delivered to students in the assignments that they were not responsible for test design.

6.1.6 Experimental Design

Considering the selection of one independent variable with two treatments, we chose the *paired comparison* design (WOHLIN *et al.*, 2012). However, to avoid the learning effect over results, we used two different experimental objects for each subject.

Subjects were divided into two groups randomly. Table 34 shows the experimental design and how subjects were assigned to experimental objects (assignments 2A and 2B) and treatments (IT and ST).

Table 34 – Experimental design

	Instructor-provided Test Cases (IT)	**Student-written Test Cases** (ST)
Assignment 2A	Group 1	Group 2
Assignment 2B	Group 2	Group 1

6.1.7 Results

We analyzed students' programs in order to calculate the correctness of each one. We used the CUnit[1] testing framework in order to execute and evaluate test results.

Individual correctness values for both testing approaches are given in figures 31 and 32. It is interesting to note that some students kept their programming performance somewhat at the same level for both approaches. Most of them (S1, S3, S6, S8, S9, S11) achieved good results for both.

Other aspect from the individual results that draws attention is that many subjects (S2, S4, S7, S10) achieved better results with student-written test cases. Only subject S5 had a considerably lower result with student-written test cases.

---

[1] cunit.sourceforge.net

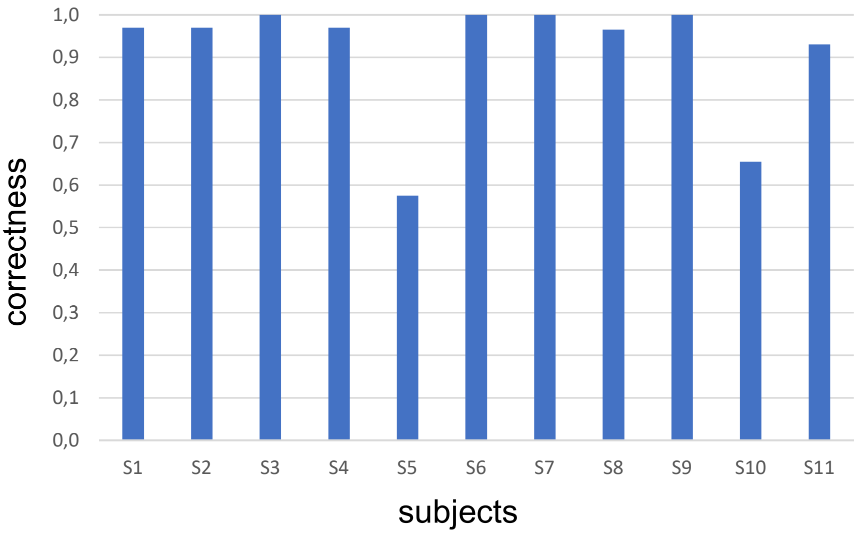


Figure 31 – Individual results for student-written test cases (ST)

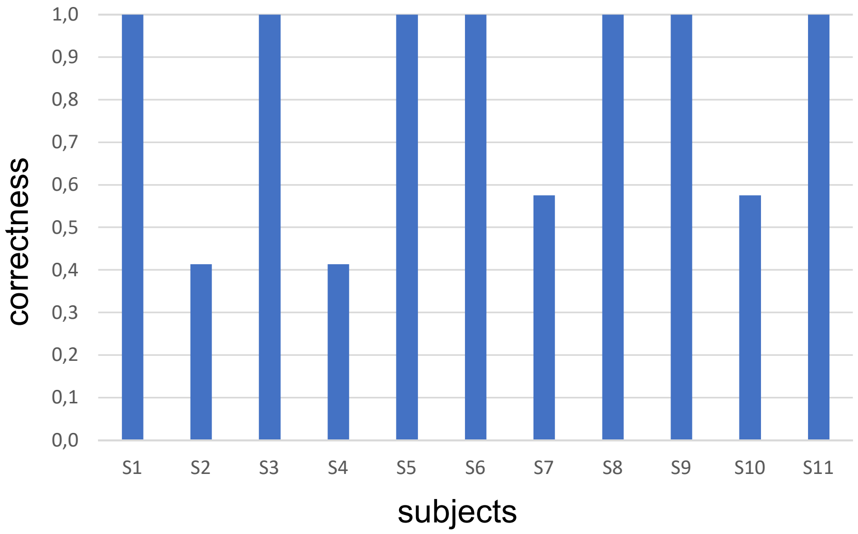


Figure 32 – Individual results for instructor-provided test cases (IT)

These results are summarized in Table 35, using the means and standard deviations for each approach. Table 36 shows results separately for each group. Comparing the means, students that wrote their own test cases (ST approach) achieved better correctness scores.

Table 35 – Mean and standard deviation for correctness

	Instructor-provided Test Cases	**Student-written Test Cases**
Mean	81.63%	91.24%
Std Dev.	25.99%	14.94%

However, these results cannot support that there is indeed a difference between the testing approaches. We applied the Wilcoxon signed-rank test, but there was not enough evidence to

Table 36 – Mean for correctness separated by groups

	Instructor-provided Test Cases	**Student-written Test Cases**
Group 1	76.55%	89.70%
Group 2	85.86%	92.53%

reject the null hypothesis at $\alpha = 0.05$ significance level. This outcome is very likely related to the small size sample.

6.1.8 Survey

As the last activity of the short course we applied a feedback questionnaire about students' experience in completing the assignments with the aid of software testing. Students' responses (Table 37) allowed us to gather some insights about their perceptions and behavior for the proposed activities.

Table 37 – Survey responses (n=11)

Q1. *Did the test cases help you implement the solution code?*	
Yes, for both assignments	81.8% (9)
Yes, just for the assignment with instructor test cases	0% (0)
Yes, just for the assignment that I wrote the test cases	18.2% (2)
No, for both assignments	0% (0)
Q2. *While you wrote the test cases, did you apply the testing criteria?*	
Yes	90.9% (10)
No	9.1% (1)
Q3. *When did you execute the test cases?*	
Only after I have completed the solution code	45.5% (5)
During implementation, even with incomplete versions of the solution code	54.5% (6)
I did not execute the test cases	0% (0)
Q4. *Did you face difficulties while writing test cases by yourself?*	
Yes	27.2% (3)
No	72.7% (8)
Q5. *Do you intend to apply software testing in programming assignments from future computing courses?*	
Yes	100% (11)
No	0% (0)

All students agree that testing practices indeed help to work on programming assignments (Q1). Most of them (81.8%) think it helped for both testing approaches and a small portion of them (18.2%) think it helped only when they wrote the test cases themselves.

The responses also provided an overview of the actual student behavior while working on the programming assignments. $90,9\%$ stated that applied the testing criteria while writing test cases (Q2). Additionally, $72,7\%$ stated that did not face difficulties while doing so (Q4).

These numbers suggest that most students were able to understand and to apply testing criteria, including freshmen.

One of the questions (Q3) assessed students' tendency to adopt test-first or test-last approaches. Students were quite divided in this matter, with almost half of them for each approach.

We did not impose a specific order to perform programming and testing activities, but we did demonstrated examples and exercises from the course material in a test-first manner and this might have influenced students' behavior.

The last question (Q5) assessed whether students intend to incorporate testing practices as part of their strategy to complete programming assignments in the future. All of them answered positively.

In the beginning of the short course they had indicated that already performed testing in their programming assignments, but most of them were not used to apply or did not know what a testing criterion was.

6.1.9 Discussion

In this investigation we were able to analyze both students' performance and perceptions of testing practices in programming assignments. Experiment results suggest that making students responsible for writing their own test cases benefit their programming performance.

Although results were not statistically significant, it is interesting to note that students' perceptions matched with experiment results. All students agree that elaborating test cases indeed helped the programming activity (Q1 in Table 37). Most of them think that working with instructor test cases also helps solving programming assignments (81.8%).

This positive effect on programming performance can be due to the fact that test design helps students better understand the problem to be solved in the assignment (FIDGE; HOGAN; LISTER, 2013). In order to select input values for test cases, they need to carefully analyze the problem domain, what forces them to think more critically about the assignment.

Regarding students' actual behavior to complete the assignments (in contrast with what they were asked to do), all students executed test cases while solving the assignment (Q3 in Table 37). Besides, almost all of them applied the testing criteria when they were supposed to write their own test cases (Q2 in Table 37).

Other interesting finding is that most of the subjects performed well with both test design approaches and some of them were able to improve their performance significantly by designing testing cases (figures 32 and 31).

These two kinds of effect could simply be related to the characteristics of the own subjects, which can have different aptitudes. However, the test design task could be considered as a possibility to help turn ineffective novice programmers into effective ones (EDWARDS *et al.*, 2009; CARTER *et al.*, 2010).

Most of the subjects were freshman or sophomore (54,54% – Table 29), and most of them were able to apply the testing criteria without major difficulties (72.7% – Table 37). This result suggests that the material about testing criteria had a difficulty level that most of them were able to follow. However, there were attending students from all levels, and there is the need to evaluate this issue with a more homogeneous sample.

In general, results show that students recognized the importance of designing test cases and that they can do it without major difficulties, even in the first-year level. This was an interesting result considering that software testing is not an easy subject, especially for freshman students. However, since we used a simple approach to represent test cases and to select input values, students were able to learn and apply it.

6.2 Experiment on students' test design skills

We conducted an experiment during the PhD student visit at the University of Alabama, which was an opportunity to collect data from a different institutional context. It was an interesting experience because of the UA institutional policies for conducting studies with human participants. Before any kind of interaction with the potential participants can happen, a study protocol must be reviewed and approved by the UA Institutional Review Board (IRB)[2]. We got the IRB protocol approved for this study on November 28th (protocol ID 17-08-462). Additionally, we intend to publish results of this experiment in:

- Scatalon, L.P.; Carver, J.; Garcia, R.E. and Barbosa, E.F. **Investigating students' software testing skills and misconceptions**. In *51st ACM Technical Symposium on Computing Science Education*. SIGCSE'20. USA.

Study Goals

Students face many difficulties to write test suites. According to Carver and Kraft's results, even senior CS students cannot thoroughly test a simple program without the help of testing tools (CARVER; KRAFT, 2011). Specially during introductory programming courses, it is possible to observe some unhelpful student behaviors, such as happy path testing (writing test suites that cover only common behavior, leaving out important parts of the program being tested) (EDWARDS; SHAMS, 2014b) and trial-and-error testing (creating unnecessarily long test suites) (CARVER; KRAFT, 2011).

These problems indicate that students have poor test design skills, since they are not able to choose test cases that would effectively test their programs. A well-designed test suite

---

[2] <http://osp.ua.edu/site/irb.html>

would cover appropriately the input space avoiding redundancy at the same time (AMMANN; OFFUTT, 2016).

When students do not design test cases systematically, they need to rely on the feedback given by testing tools, such as test coverage, to adjust their test suite. The information about test coverage may be useful to improve the current test suite, but it does not improve the tester mindset of the student and her/his test design skills.

Coverage tools provide *what* program elements have not been covered, which can motivate students to perform trial-and-error testing until they find test cases that cover them. On the other hand, testing techniques and criteria would help students to perform systematic testing, knowing *how* to write a proper test suite and understanding *why* it is so.

In this scenario, we aim to investigate the development of students' test design skills as they learn fundamental testing concepts, such as coverage criteria. Also, we intend to explore *why* students' test suites are incomplete in terms of testing concepts. In this way, it is possible to point out which concepts they have difficulties with.

Therefore, this study focuses on the following research questions:

- **RQ1:** Are students able to improve their test design skills as they learn software testing concepts?
- **RQ2:** Can students better use feedback on test coverage to improve their test suites as they learn software testing concepts?
- **RQ3:** What misconceptions about software testing can be observed in students' test suites?

Hypotheses

The research questions RQ1 and RQ2 gave rise to two hypotheses we intend to investigate:

- **H1**: Students will be able to progressively improve their test design skills as they learn fundamental testing concepts.
- **H2**: Students will improve their test suites more systematically after learning fundamental testing concepts.

Context

We conducted the study in a Software Testing course (CS 416/516) at the University of Alabama. Course topics included techniques and tools for software testing and quality assurance. The textbook used was "Introduction to Software Testing" (2nd edition) by Ammann and Offutt (AMMANN; OFFUTT, 2016). We chose this course because it covers basic software testing concepts, and we would be able to keep track of the evolution of students' test design skills as they learn different coverage criteria.

Artifacts

Students tested four short Java programs (˜ 50 SLOC each) during the study:

- **Vending Machine (P1)**: a simplified representation of a vending machine, from (ORSO *et al.*, 2001).
- **Identifier (P2)**: a program that determines if a given identifier is valid or not, from (VINCENZI *et al.*, 2010).
- **Five Bit Encoding (P3)**: a simple string compression algorithm using a 5-bit encoding system, from (BANDARA, 2011).
- **Insulin Pump System (P4)**: the dose controller of an insulin pump system, from (SOMMERVILLE, 2010).

Variables

In order to capture the effect of learning software testing concepts in different stages, the independent variable was **course modules**. We focused in the modules of the basic concepts and of two coverage criteria. The variable had four cumulative levels: *foundations*, *input space partitioning*, *graph coverage* and *other modules*.

The outcome that we were interested in is students' test design skills. According to Ammann and Offutt, test design involves choosing test cases that compose a test suite, in such a way that they cover appropriately the input space avoiding redundancy at the same time (AMMANN; OFFUTT, 2016). Therefore, we intended to check if these characteristics were reflected in their test suites.

In this sense, the dependent variables of the study were *test coverage* and *test redundancy*, both measured by running students' test suites against the programs they were supposed to test. Test coverage is a common way to assess students' test cases. However, test redundancy is less explored, despite being an important part of test design.

In this study, **test coverage** was measured by *statement coverage*, *branch coverage* and *condition coverage*. **Test redundancy** is the rate of test cases that cover the same program elements (statements, branches and conditions), being measured by:

$$redundancy = \frac{\#redundant\ test\ cases}{\#submitted\ test\ cases}$$

Procedure

We collected data from students' submissions for homework assignments. After each course module, students were supposed to complete a homework, with the purpose of practicing a different coverage criterion.

For all homework assignments, students were asked to write "the smallest, yet most complete" JUnit test suite for a given program. Additionally, we asked them to provide a brief explanation of why each test case was added in the test suite.

Students were supposed to apply different coverage criteria to test the programs:

- **Homework 1**: ad hoc testing on P1, with no specific criterion.
- **Homework 2**: testing on P2, with input space partitioning.
- **Homework 3**: testing on P3, with graph coverage.
- **Homework 4**: testing on P4, with both input space partitioning and graph coverage.

The study design is outlined in Figure 33. It was composed by two steps, similarly to the study of Carver and Kraft (CARVER; KRAFT, 2011): (1) creation of JUnit test suite (for all assignments) and (2) update of test suite based on the received feedback about test coverage (for assignments homework 1 and homework 4).

For the purposes of assessment, all assignments were graded based on how well students followed the instructions and whether they made a real attempt at completing the assignment. Even so, students got feedback after each assignment about the test coverage they achieved and, when was the case, of their coverage criteria application.

After the course ended, the PhD student collected the homework submissions of consenting students and performed data anonymization. Currently, we are performing the data analysis, which involves using the tool CodeCover[3] to calculate students' test coverage and redundancy. Additionally, we are working on a qualitative analysis of the reasons that students indicated for each test case.

We will write a paper with the experiment findings and also intend to submit it soon:

- Scatalon, L.P.; Carver, J.; Garcia, R.E. and Barbosa, E.F. **Investigating students' software testing skills and misconceptions**. *In SIGCSE Technical Symposium (SIGCSE 2020).*

---

[3] <http://codecover.org>

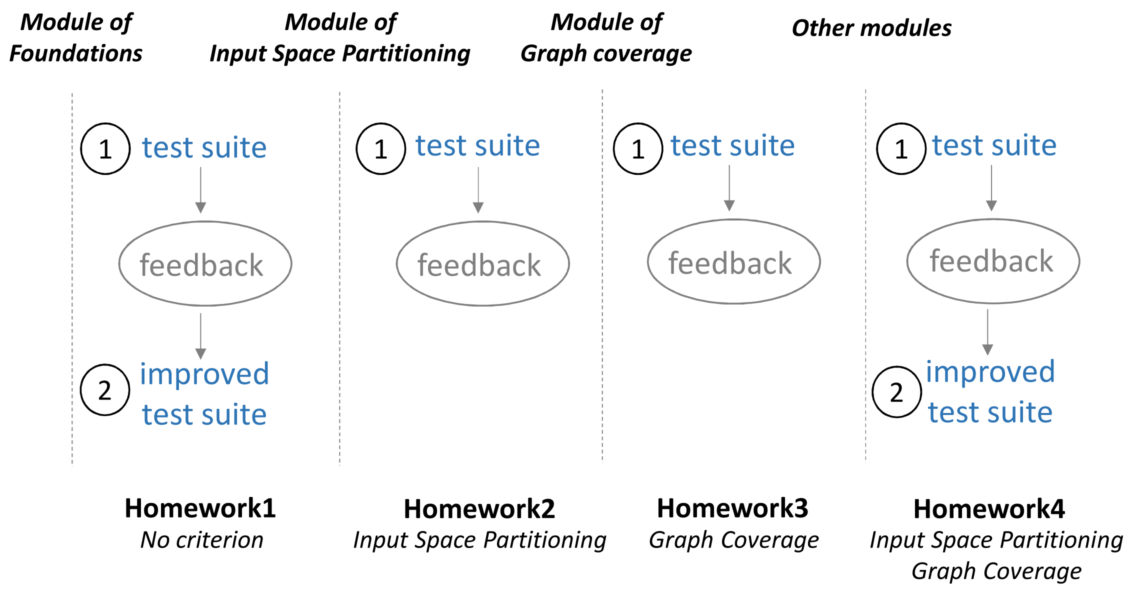


Figure 33 – Study design

6.3 Final remarks

In this chapter we discussed two experiments conducted during this PhD work. The first one investigated the test design task during the process of working on programming assignments. In studies similar to our investigation, the testing activity is usually seen from a holistic point of view. Conversely, in our study we decomposed the testing activity into individual tasks and investigated specifically the effect of the test design task.

We conducted an experiment and a survey with students that participated in a short course we offered about this subject. Results suggest that students' programming performance can be enhanced when they write their own test cases. Moreover, students' perceptions matched experiment results. Students recognized the relevance of software testing as a supporting practice when completing the proposed assignments.

However, in order to enable students to write and improve their test suite, they need to be instructed on testing concepts. They need to learn how to select input values that will fully and effectively test their program. This issue motivated us to instruct students on two testing criteria, *equivalence partitioning* and *boundary value analysis*. In this way, they were able to choose input values systematically, instead of adopting a trial-and-error approach (CARVER; KRAFT, 2011).

This experience showed the importance of instructing students on how to select appropriate input values and then elaborate test cases. They need background knowledge to perform test design if they are going to be responsible for it.

CHAPTER 7

EXPERIMENTAL FRAMEWORK FOR THE INTEGRATION OF SOFTWARE TESTING INTO PROGRAMMING EDUCATION

The teaching of programming in higher education is a context with a lot of variability. There are different institutional contexts and target audiences across STEM disciplines. Also, programming concepts can be taught using different programming paradigms, languages, platforms and practices. Hence, there are many different approaches to teach programming. In this scenario, empirical studies help to understand how and when a given approach would yield better learning outcomes for students from different contexts.

However, empiricism in programming education (and in Computer Science education, in general) suffers from a lack of research rigor (FINCHER; PETRE, 2004; PEARS; MALMI, 2009; ROBINS, 2015; LISHINSKI *et al.*, 2016), mainly in terms of the high number of experience reports and a lack of theoretical basis to design studies and discuss the obtained results (FINCHER; PETRE, 2004; VALENTINE, 2004; RANDOLPH, 2007; SHEARD *et al.*, 2009; MALMI *et al.*, 2010; MALMI *et al.*, 2014; AL-ZUBIDY *et al.*, 2016). In this sense, domain-specific models to scope and design experiments can help to deal with these problems. In particular, an experimental framework provides a structure that supports researchers to design future studies and also to frame existing studies in order to make the research limits clear, in terms of domain concepts.

In this chapter we discuss the creation of the proposed **E**xperimental **F**ramework for the integration of **S**oftware **Te**sting into **P**rogramming education (STeP-EF). We chose the integration of software testing as the teaching approach to be explored in our framework due to its prominence in the literature and specially because of its importance for computing undergraduate programs. In Section 7.1 we explain our method to build the experimental framework. The following sections present the three models that compose STeP-EF: the goal model in Section 7.2 and the variable model in Section 7.3. Additionally, we demonstrate how the experiments described in Chapter 6 fit in the proposed framework.

7.1 Experimental framework building method

In order to build the proposed experimental framework, we considered the methods used by researchers that have established this kind of framework in the literature (see Section 3.4). It is possible to notice the same sources of information used by researchers to compose the frameworks: (i) existing studies in the literature of the research domain and (ii) studies conducted by the researchers themselves.

So, we followed the same approach: (i) we conducted a systematic mapping of the literature in order to obtain the elements to compose the framework (see Chapter 5) and (ii) we conducted experiments within the scope of the domain of interest, which is the integration of software testing into programming education (see Chapter 6).

The systematic mapping data extraction provided fragmented elements, which we assembled in the result analysis, already aiming to build the proposed framework. Basically, we clustered similar elements (i.e. the variables extracted from the empirical studies) and organized them according to a common structure, using the concepts of the Software Testing area as a reference.

For example, we noticed that in some studies students were responsible for writing their own test cases, and in others they received ready-made test cases. This situation led us characterize it in terms of the testing tasks that compose the testing activity, according to Ammann and Offutt (2016): test design, test automation, test execution and results evaluation. Then, it is possible to describe more clearly how the testing practice is configured in the classroom, in terms of testing concepts. When students receive instructor test cases, they conduct the tasks of test execution and results evaluation. Otherwise, they are responsible for all four testing tasks. Note that this description helps to better understand what the adopted testing practice is and what is not.

The experimental framework, entitled STeP-EF (**E**xperimental **F**ramework for the integration of **S**oftware **Te**sting into **P**rogramming education), is composed by three models:

- **STeP-EF goal model**: an instantiable goal, similar to the framework on software reading techniques of Basili, Shull and Lanubile (1999).

- **STeP-EF variables model**: a model of independent, dependent and context variables, similar to the framework on pair programming of Gallis, Arisholm and Dyba (2003). The model of dependent variables also include a model of metrics, similar to the framework on XP practices of Williams *et al.* (2004).

7.2 STeP-EF goal model

This model consists in an instantiable goal, which is supposed to help the researcher during the scoping phase of the experimental process. We used the goal template from the GQM model and completed it with general values, in order to allow the instantiation of specific goals for experiments in the domain of the integration of software testing into programming education.

There are five parameters in the GQM goal template (see Section 3.3.1): *object of study*, *purpose*, *focus*, *point of view* and *context*.

Analyze <*Object(s) of study*>
for the purpose of <*Purpose*>
with respect to their <*Focus*>
from the point of view of the <*Perspective*>
in the context of <*Context*>

According to Basili, Shull and Lanubile (1999), the **purpose** depends on the researcher's intent in conducting the study, which could be to *characterize*, *monitor*, *evaluate*, *predict*, *control* or *change* some characteristic in the object of study. The **point of view** is the perspective from which the data is analyzed, which could be the perspective of the *researcher* or the *instructor*, for example.

The remaining parameters (*object of study*, *focus* and *context*) are the domain-specific elements that compose the goal. These parameters are part of the experimental framework and, therefore, are going to be characterized by its domain-specific models.

The **object of study** in this particular domain is the **teaching method** used to integrate software testing into programming education. Teaching methods are the different ways to teach programming with the integration of software testing. They can be characterized by how *testing concepts* are integrated into **course materials** and how *testing practices* are integrated into **programming assignments**. Also, **supporting tools** can be used with the purpose to teach testing concepts in this context or to automate some aspect of testing practices in programming assignments.

In other words, the teaching method can be characterized by how testing is incorporated into the materials and assignments of the programming course and by the supporting tools used to ease any aspect of this incorporation. For example, a teaching method which is more clearly delineated in the literature is *Test-Driven Learning* (TDL) by Janzen and Saiedian (2006a), Janzen and Saiedian (2008). The integration of testing into programming courses as proposed in TDL could be structured as:

- *Course materials:* TDL proposes that code examples including unit tests should be used during lectures.

- *Programming assignments:* TDL encourages the use of test-first programming (TDD) as the process to develop assignments' solutions, but it can also be implemented using test-last programming.
- *Tools:* The authors use a tutor system called Web-IDE and also indicate using a xUnit testing framework if students have enough maturity, otherwise they recommend the assert command, given its simplicity.

The **focus** is the effect of the object of study which is of interest in the study, i.e. the focus of the investigation. The ultimate focus of a study on programming education is to evaluate students' **learning outcomes**. In this sense, besides using grades to do so, the outcomes of a teaching method can be observed through students' programs and tests, their performance and behavior while completing assignments, and also their attitudes towards the teaching method.

The **context** is a description of the study environment. In this domain, it is the **programming education context** where the study is conducted. It is characterized by the students participating in the study and the respective programming course where it takes place.

Considering these general values, the **goal model** can be represented as:

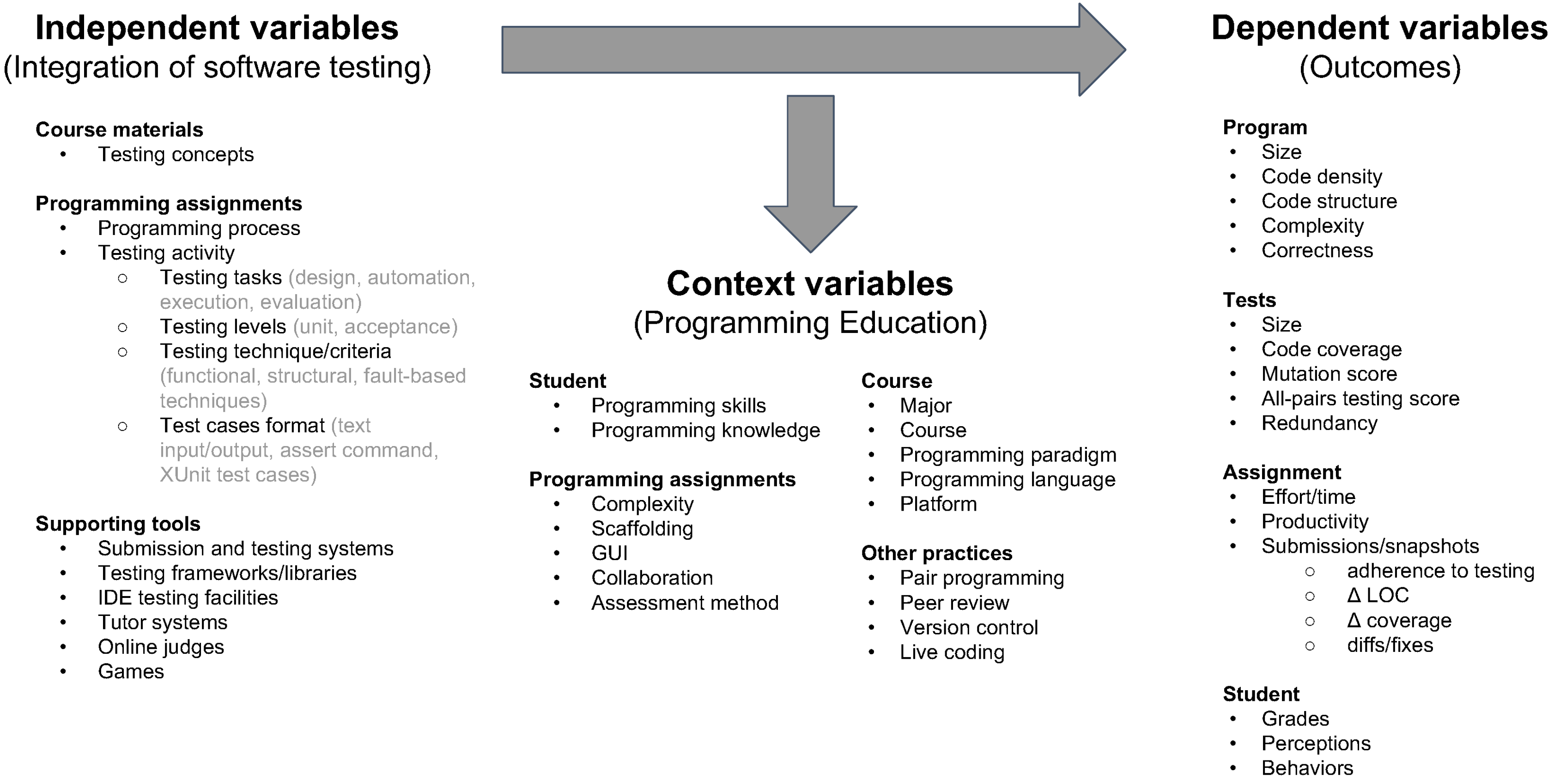
Analyze ***teaching method*** *to integrate software testing*
for the purpose to <*purpose*>
with respect to ***learning outcomes***
from the point of view of the <*perspective*>
in the context of ***<programming education context>***

The model of variables (discussed in the next section) should help to model these general values (i.e. *teaching methods*, *learning outcomes* and *programming education context*) in terms of their characteristics, which can be selected as variables in a study.

7.3 STeP-EF variables model

The model depicted in Figure 34 consists in a catalog of variables from the domain of the integration of software testing into programming education. The idea is to support the researcher in the planning phase of the experimental process, more specifically in the variable selection activity.

This model helps to refine the domain-specific elements established in the experiment goal. Linking both models, *independent variables* represent characteristics of the *object of study*, *dependent variables* compose the *focus* of the investigation (they are the results chosen to be observed among the many possibilities of results to observe) and *context variables* are used to describe the experimental *context*.



Independent variables
(Integration of software testing)

Course materials
- Testing concepts

Programming assignments
- Programming process
- Testing activity
    - Testing tasks (design, automation, execution, evaluation)
    - Testing levels (unit, acceptance)
    - Testing technique/criteria (functional, structural, fault-based techniques)
    - Test cases format (text input/output, assert command, XUnit test cases)

Supporting tools
- Submission and testing systems
- Testing frameworks/libraries
- IDE testing facilities
- Tutor systems
- Online judges
- Games

Context variables
(Programming Education)

Student
- Programming skills
- Programming knowledge

Programming assignments
- Complexity
- Scaffolding
- GUI
- Collaboration
- Assessment method

Course
- Major
- Course
- Programming paradigm
- Programming language
- Platform

Other practices
- Pair programming
- Peer review
- Version control
- Live coding

Dependent variables
(Outcomes)

Program
- Size
- Code density
- Code structure
- Complexity
- Correctness

Tests
- Size
- Code coverage
- Mutation score
- All-pairs testing score
- Redundancy

Assignment
- Effort/time
- Productivity
- Submissions/snapshots
    - adherence to testing
    - Δ LOC
    - Δ coverage
    - diffs/fixes

Student
- Grades
- Perceptions
- Behaviors

Figure 34 – STeP-EF variables

7.3.1 Independent variables

The model of independent variables (on the left side of Figure 34) contains a characterization of possible input variables in the domain. Therefore, it can serve as a basis for selecting independent variables that should be manipulated during the study, parameters that should be kept constant and, if it is the case, blocking variables.

The activity of variable selection can be greatly facilitated by having an overview of input variables available, mainly because all the characteristics of the teaching method should be taken into account, even when not selected as independent variable, the remaining input variables should be kept constant, so the researcher must be aware of them. Otherwise, unknown variables can cause undesired and unaccounted for systematic effects, also known as confounding factors.

As pointed out by our systematic mapping results (in Chapter 5), very few selected studies explicitly indicate an independent variable or discuss the variable selection rationale. Nevertheless, most studies provided a characterization of the underlying teaching method used to integrate software testing. Hence, we discuss such characteristics extracted from studies in the next subsections.

7.3.1.1 Course materials

The integration of software testing into a programming course also involves the teaching of testing concepts in this context. In other words, it is the theoretical part of such integration. In this sense, there is the need to understand how testing and programming concepts should be combined together in the course materials, i.e. which programming concepts are prerequisites to learn which testing concepts and practices. Barbosa *et al.* (2008) and Agarwal, Edwards and Perez-Quinones (2006) proposed models in this direction.

However, considering the results of the systematic mapping, there are no empirical studies providing evidence regarding this aspect of combining programming and testing concepts. There are only studies investigating the effect on students' programming performance with and without knowledge of testing concepts, i.e. before and after learning testing concepts (LEMOS *et al.*, 2015; LEMOS *et al.*, 2017) and with different levels of computing knowledge, i.e. during different courses across the computing program (GóMEZ; VEGAS; JURISTO, 2016).

Focusing on another aspect, Desai, Janzen and Clements (2009) investigated the adaptation of existing programming course materials to accommodate testing practices. They evaluated the effects in terms of staff and student work loads.

Still, considering studies whose focus was not the testing concepts, for some of them researchers also describe the teaching of testing concepts to students. However, as indicated by the systematic mapping results, it is often not clear what and how testing concepts were taught.

7.3.1.2 Programming assignments

The integration of testing practices into programming assignments is the most explored aspect in this domain. The idea is to investigate how the activities of programming and testing can be combined in this context, considering the background of novice programmers. In this sense, a **programming process** can be used to provide guidance to students on how they should conducted both activities.

Experiments that focus on the programming process (or development approach) usually compare different approaches for the adopted process in the classroom. For example, some studies compare test-first programming (TDD) with test-last programming (and even ad hoc programming) (ERDOGMUS; MORISIO; TORCHIANO, 2005; JANZEN; SAIEDIAN, 2006a; JANZEN; SAIEDIAN, 2007; PARODI *et al.*, 2016). Other possible configuration is to compare a given programming process with another control approach, such as the “traditional” approach used in previous offerings of the course before the introduction of the investigated process or ad hoc programming (NETO *et al.*, 2013; PARODI *et al.*, 2016).

There are also studies that focus on different configurations of the **testing activity**. In particular, there are two studies on the test design task. The first compares TDD and TDD with testing criteria (CAMARA; SILVA, 2016), which can be characterized as comparing ad hoc student test design and test design using testing criteria. The second is a study conducted by us during this PhD work, comparing the test activity conducted with instructor-provided test cases and student-written test cases (SCATALON *et al.*, 2017b), i.e. students conducting the test design task versus not conducting it.

7.3.1.3 Supporting tools

Tools can be used to provide support both to the integration of testing concepts into course materials and the integration of testing practices into programming assignments. Considering the the categories of tools that we identified in the systematic mapping, *tutor systems* can be used to teach testing concepts and *testing frameworks/libraries*, *IDEs’ testing facilities*, *submission and testing systems*, *online judges* and *games* can be used to automate different aspects of the testing practice adopted in the classroom.

Still considering the systematic mapping results, most papers related to tools in this context were just proposals without an evaluation (mapped to the category "Not Applicable") and many were experience reports or case studies, what means that many studies involving tools do not present a manipulation of independent variables. Even for studies that do present an indication of variable manipulation, the papers do not mention explicitly what is the independent variable under investigation. Nonetheless, we considered different groups of results as treatments investigated in the experiments.

In this sense, some treatments/levels configurations are repeated across papers which investigated tools in this context by means of experiments. The most common configuration is the comparison **with tool / without tool** (DALY; HORGAN, 2004; THORNTON *et al.*, 2008; DVORNIK *et al.*, 2011; WANG *et al.*, 2011; BUFFARDI; EDWARDS, 2013b; JANZEN; CLEMENTS; HILTON, 2013; VUJOSEVIC-JANICIC *et al.*, 2013; ALLEVATO; EDWARDS, 2014; REYNOLDS *et al.*, 2015). The treatment *without tool* usually means the "traditional" approach without the use of the investigated tool to conduct some teaching activity. This treatment may hold values, for example, of previous offerings of the course in which the investigated tool was not used yet.

The second configuration is when the study compares **different versions of the same tool** (version 1 / version 2 / ... / version n) (JEZEK; MALOHLAVA; POP, 2013; BUFFARDI; EDWARDS, 2014b; BUFFARDI; EDWARDS, 2014a; BRAUGHT; MIDKIFF, 2016). The different versions may include different functionalities, what means different kinds of support for the same teaching activity. Finally, there is also the possibility of comparing the investigated tool with another tool usually employed for the same activity (**with tool / with similar tool**) (BLAHETA, 2015).

7.3.2 Dependent variables

The model of dependent variables (on the right side of Figure 34) contains possible output variables in the domain. The dependent variable represent the effect of the teaching method, i.e. the results of its application in the classroom. The primary effect to be observed is students' learning. However, this effect is not directly measurable, so other variables are employed to represent it.

In this sense, we organized the variables gathered in the systematic mapping and sorted them by entity being measured. Hence, these entities are the source of measures used to gauge students' learning on programming and testing in this context. We identified four entities (program, tests, assignment and students), which are described in the following subsections.

7.3.2.1 Program

Students' submitted programs can provide measures to evaluate their programming skills, which indicates whether they are able to apply programming concepts to implement a solution.

- **Size**: the size of students' programs is commonly measured in terms of lines of code (LOC) or other variants such as non-comment lines of code (NCLOC). This metric is often used to other indirect metrics such as students' productivity while completing the programming assignment.

- **Density**: the code density put the program size into perspective with its structure. Possible metrics include LOC/method, LOC/class, LOC/feature.

- **Structure**: the structure provides indication of the internal quality of students' programs (JANZEN; SAIEDIAN, 2006a). Structural metrics include number of classes, number of parameters, coupling between objects, among others that can be obtained through tools such as Metrics[1] and CCCC[2]. Other important structural metric commonly used is the cyclomatic complexity.

- **Style**: style metrics are related to the readability of the program. In this sense, a static analysis tool, such as Checkstyle[3] can indicate whether students' programs are adhering to coding standards (naming conventions, code layout, documentation etc).

- **Correctness**: correctness is supposed to indicate how correct the student program is, according to the problem specification. In this sense, correctness is measured by means of the pass/fail rates of a test suite that represent the expected program behavior. This test suite can be composed of instructor test cases or the set of test cases developed from all students in a given class (which comprises the approach of all-pairs testing).

7.3.2.2 Tests

When students are supposed to submit test cases to complete assignments, their tests can also provide metrics, which indicate their testing skills.

- **Size**: Similarly to program size, it is also measured by LOC or NLOC, but usually called test LOC. Other metrics include number of asserts and number of test cases.

- **Code coverage**: Code coverage is the most common metric of test quality. It expresses the percentage of code elements that are exercised (covered) by the test suite. Different code elements can be considered, which raises different coverage metrics, such as line coverage, statement coverage, branch coverage, condition coverage, among others (SHAMS; EDWARDS, 2015). The idea is that the test suite should exercise as many code elements as possible in order to reveal the presence of defects. The main drawback in using code coverage for assessing students' tests is that they can achieve 100% coverage in some situations that do not necessarily mean they have a good test suite. Their test suite may not be checking for missing features in their submitted programs, for example. This is a cause for concern, specially considering that students tests might be mainly covering mainstream behavior (EDWARDS; SHAMS, 2014b)

- **Mutation score**: Mutation analysis includes the generation of several defective programs, the so-called "mutants", and the execution of the test suite against all of them. The mutation score is the rate of mutants "killed" by the student test suite, i.e. mutants that presented a test result different from the original program for a given test case.

---

[1] <http://metrics.sourceforge.net/>

[2] <http://cccc.sourceforge.net/>

[3] <http://checkstyle.sourceforge.net/>

- **All-pairs testing score**: All-pairs testing involves executing each submitted program against all submitted test suites and vice versa. Edwards and Shams (2014a) propose the calculation of the all-pairs testing score as the percentage of students' programs in the class that failed at least one test case in the test suite being evaluated.
- **Redundancy**: Test redundancy is the rate of test cases that cover the same program elements (e.g. statements, branches and conditions), being measured by: $redundancy = \#redundant\ test\ cases / \#submitted\ test\ cases$

7.3.2.3 Assignment

The variables on the **assignment** are related to the students' process to develop program and tests and to the assignment assessment.

- **Effort/time**: Effort in a development team is measured in terms of the amount of people involved in a given activity and the time they took to complete it (person-hours). However, in the studies from this context, effort is usually measured for each individual student, in terms of the time they took to complete the assignment.
- **Productivity**: Productivity is measured by the amount of code produced divided by the effort required to produce it, i.e. size/effort.
- **Submissions/snapshots**: When a submission system is used and multiple submissions are allowed, there is a possibility to follow students progression throughout the process of completing the assignment. Each submission becomes a snapshot of students' work (or the system itself collects code snapshots automatically). In this way, it is possible to collect a sequence of metrics of students' work, and produce other process-related metrics.
  - **Δ LOC**: The difference in the amount of lines of code between two submissions. It can be calculated from two successive submissions or between the first and last one, for example (BUFFARDI; EDWARDS, 2014a).
  - **Δ Coverage**: The difference in the the results of code coverage between two submissions of students' tests for a given programming assignment (BUFFARDI; EDWARDS, 2014a).
  - **Adherence to testing**: It is possible to evaluate whether students are indeed testing their own programs, by calculating the ratio of the size of their tests (e.g. test LOC) and the size of their programs (e.g. program LOC) (BUFFARDI; EDWARDS, 2012b; BUFFARDI; EDWARDS, 2013a; BRAUGHT; MIDKIFF, 2016; BUFFARDI; EDWARDS, 2013b).
  - **Diffs/fixes**: Diffs are edits performed by students in the source code, computed from two successive snapshots. These diffs can be analyzed to find out which code

elements students changed, either considering low-level changes, such as statement and expression fixes (SOUZA; KOLLING; BARBOSA, 2017), or computing student progress indicators, such as adding new method, removing static analysis errors, reducing cyclomatic complexity etc (EDWARDS; LI, 2016).

7.3.2.4 Student

The variables centered on the **student** are the ones inherently related to students, aiming to evaluate concept understanding and attitudes towards the teaching methods.

- **Grades**: Investigating students' knowledge/concept understanding of a given subject usually involves analyzing their obtained grades in a quiz, test or exam. The overall grades in the corresponding course are often used too. The results validity is increased if researchers use a validated assessment instrument, such as Utting *et al.* (2013b).
- **Perceptions/Behaviors**: Students' perceptions indicate their opinions about the testing approaches. Students' behaviors refer to what they actually do during programming assignments, in contrast with what they were instructed to do. Both are usually investigated through surveys, by asking students about their opinions and what they actually did during assignments (JANZEN; SAIEDIAN, 2007).

7.3.3 Context variables

The model of context variables (in the center of Figure 34) lists variables from the context of programming education. These variables are not about the testing practices or concepts per se, but may affect the results of a teaching method involving the integration of software testing. We discuss these variables in the next subsections, sorted by what aspect of the context they represent (student, programming assignments, course, other practices).

7.3.3.1 Student

Some variables are related to the students' background, especially to gauge students' current **programming skills** and **knowledge** (JANZEN; SAIEDIAN, 2007; GóMEZ; VEGAS; JURISTO, 2016). Students present a different level of programming skills in each programming course. Therefore, there is the need to assess whether students' programming skills are adequate to conduct the testing practice in question or to learn the involved testing concepts. In this sense, the adopted testing practices can present different levels of difficulty to match the students' programmming background (JONES, 2001).

7.3.3.2 Programming assignments

There are some characteristics of programming assignments that are not directly related to the testing practice adopted, but could influence its aplication.

- **Complexity**: Programming assignments can present different levels of complexity across computing courses. Even when considering the same course, the complexity varies from simple programming assignments to more elaborate programming projects, for example. When the programming problem is too simple, students may not perceive the benefits of conducting the testing practice (JANZEN; SAIEDIAN, 2007; NETO *et al.*, 2013; BUFFARDI; EDWARDS, 2013a).

- **Scaffolding**: Instructors can provide additional resources to students along with the assignment description, such as a skeleton/template code to help them start working on their solution (SPACCO; PUGH, 2006; UTTING *et al.*, 2013b; TEUSNER; HILLE; HAGEDORN, 2017). This issue is directly related to the program design freedom that students will have in their solutions, since a skeleton code probably includes the methods' signature (or functions' prototypes). The program design freedom affects the testing practice in the sense that, if instructors' tests are used by students, their programs should have a fixed design in order to compile properly with the tests (SHAMS, 2013a; SHAMS; EDWARDS, 2013; FIDGE; HOGAN; LISTER, 2013; BUFFARDI; EDWARDS, 2014a).

- **GUI**: When the programming assignment involves a graphical interface, students should also consider the testing of graphical elements in their programs. However, they may face difficulties to properly test GUI-based assignments (THORNTON *et al.*, 2008; FIDGE; HOGAN; LISTER, 2013).

- **Assessment method**: The assessment method, whether automatic or manual (PIETERSE; LIEBENBERG, 2017), and the assessment criteria can influence the teaching method as well. Automatic assessment provides a natural context to students conduct software testing, at least the test results evaluation. Other issue is whether students tests are going to be graded in the programming assignment.

7.3.3.3 Course

The introductory sequence is composed by several programming courses and each one provides a different context to the integration of software testing. In particular, programming concepts addressed vary and students have different previous programming experience in each course.

- **Major**: 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 paradigm and 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* (ACM/IEEE-CS, 2001)). 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) (DAVIES; POLACK-WAHL; ANEWALT, 2011).

- **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 (MARKHAM; KING, 2010; COWDEN *et al.*, 2012; EDWARDS; ALLEVATO, 2013). 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.

7.3.3.4 Other practices

Other practices may be used in combination with the integration of software testing, which have their own variables having an effect on the teaching of programming. Therefore, their interaction should be considered as well.

- **Pair programming**: Pair programming is not restricted to programming, other activities such as testing can also be paired (FIDGE; HOGAN; LISTER, 2013).
- **Peer review**: Instructors can arrange for students to review each others' programs and tests. "Reading others' tests might improve a student's abilities as a tester" (POLITZ; KRISHNAMURTHI; FISLER, 2014; POLITZ *et al.*, 2016).
- **Version control**: Version control tools are widely used in industry, so it is a good practice to introduce them earlier to novice programmers (BAUMSTARK JR.; ORSEGA, 2016; KRUSCHE; SEITZ, 2018).
- **Live coding**: Live coding consists in the instructor writing code "live" from scratch during lectures (GASPAR; LANGEVIN, 2007a; GASPAR; LANGEVIN, 2007b; RUBIN, 2013).

7.4 Instantiation of experiments into the framework

In this section we instantiate the experiments conducted during the PhD project into the experimental framework. The idea is to show how the conducted experiments fit in the framework, emphasizing the selected variables in the experimental design. We followed an approach similar to Shull et al (SHULL *et al.*, 2005), in the sense of organizing information of the experimental design from different experiments using the same structure.

So, considering the structure of Figure 34, the instantiation consists in showing what independent and dependent variables were selected and the respective treatments and metrics used in the experiment. Also, it consists in characterizing the experimental context, by indicating which values the context variables hold. Among the possible independent variables, generally one is selected as a factor in the experiment, which is the case of both experiments discussed in this section. The remaining independent variables should hold constant values, called parameters, in order to avoid producing undesired systematic effects.

The first experiment is the one described in Scatalon *et al.* (2017a) and in Section 6.1. As highlighted in blue in Figure 35, the selected independent variable is related to the **programming process**, namely the *test design task*, with two treatments: *instructor-provided test cases* and *student-written test cases*. The effect was investigated in terms of program **correctness**, measured by the *pass rate* of a reference test suite.

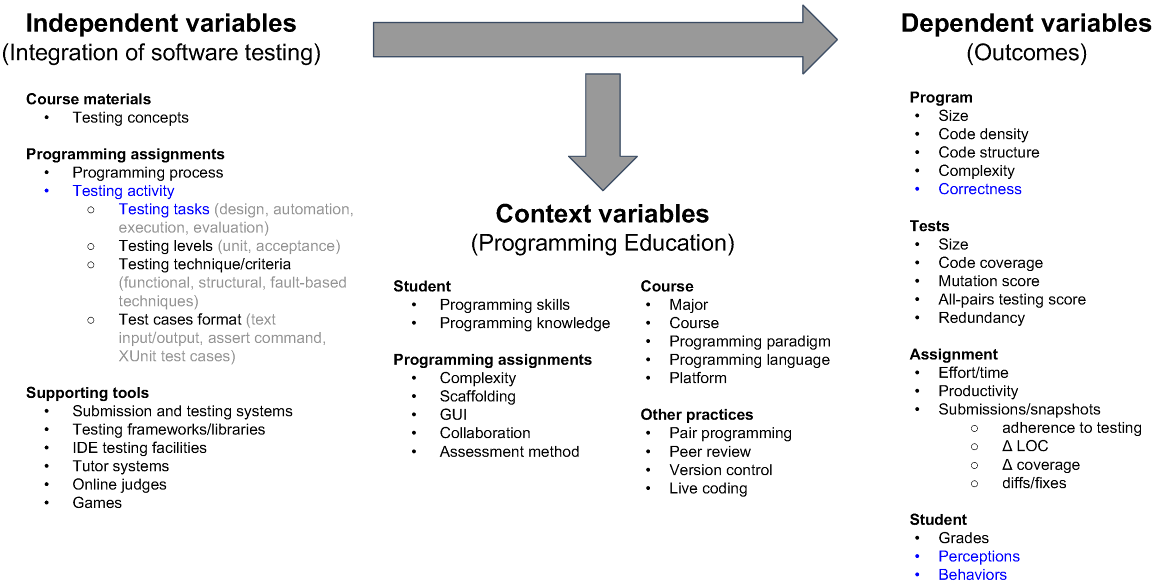


Figure 35 – Experiment described in (SCATALON *et al.*, 2017a)

Still considering independent variables (or input variables), but now focusing on the parameters, the *course material* we used had an introduction to functional testing criteria (equivalence partitioning and boundary value analysis) and automated unit testing in C (with the assert command). The *programming assignments* were about clock features, such as add and subtract time values, and we did not use any specific *tool* to support the testing practice, students executed their tests in the IDEs with which they were already familiar.

About the experimental context, students were Computer Science *majors*, taking *courses* of Data Structures I, Data Structures II and Object Oriented Programming. In addition, the experiment activities involved the imperative *programming paradigm* with the C *programming language*.

The second experiment was described in Section 6.2. Similarly, the aspects that represent the selected variables are highlighted in blue in Figure 36. The selected independent variable was the completed modules in the course material on **testing concepts**, with four cumulative levels as treatments: foundations, input space partitioning, graph coverage and other modules. The effect was investigated in terms of **code coverage** and **test redundancy**.

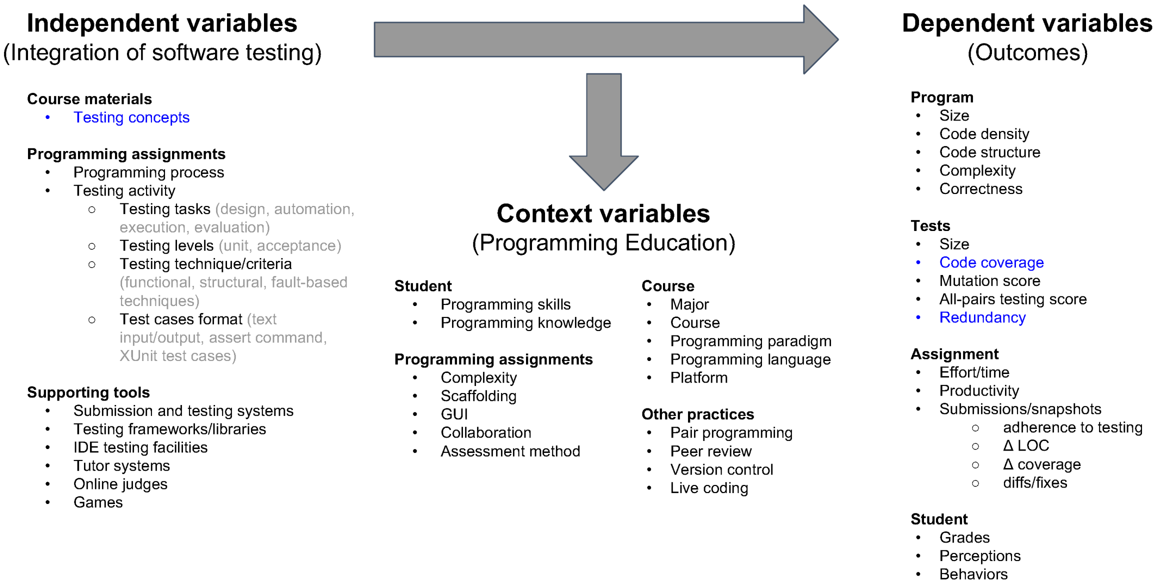


Figure 36 – Experiment described in Section 6.2

The *programming assignments* were from different domains, yet with similar complexity, namely vending machine (P1), identifier (P2), five bit encoding (P3) and insulin pump system (P4). About the *programming and testing process*, we did not recommend any specific process to students. As supporting *tool*, students used JUnit to write and execute test cases.

Regarding the experimental context, students were also Computer Science *majors* taking the course CS416 Testing and Quality Assurance. We used the Java *programming language*, and hence the object oriented *programming paradigm*.

CHAPTER 8

CONCLUSIONS

The variety of ways to address programming education in the classroom stands out in many aspects: there are different target audiences (computing majors and non-majors, for example), different ways to implement the introductory sequence curriculum, different programming languages and paradigms, different platforms (desktop/laptop, mobile) and different teaching approaches. In this scenario, experimental studies help to uncover these variables related to the teaching of programming and investigate their relationships with students' learning outcomes. In other words, empirical studies provide evidence about when and how students learn programming better.

However, empirical studies in programming education (and Computer Science Education in general) often present a lack of research rigor (FINCHER; PETRE, 2004; PEARS; MALMI, 2009; ROBINS, 2015; LISHINSKI *et al.*, 2016). In particular, many empirical studies in this area consist of experience reports (VALENTINE, 2004; RANDOLPH, 2007), which do not involve a planned data collection and the researcher (often the course instructor) provides a report of how students responded to a given approach in the classroom. In this kind of studies, the variables of the teaching of programming are not carefully identified and isolated, so researcher's claims have no guarantee (i.e. anecdotal evidence), since it is based only on opinions, instead of properly identifying causes and effects related to the teaching approaches employed in the classroom.

Besides, studies in this context also suffer from a lack of theoretical basis (BEN-ARI *et al.*, 2004; BERGLUND; DANIELS; PEARS, 2006; SHEARD *et al.*, 2009; MALMI *et al.*, 2010; KOULOURI; LAURIA; MACREDIE, 2014; MALMI *et al.*, 2014). The variables selected in a study are actually theoretical concepts/constructs in a given area. Therefore, researchers ideally should investigate in their studies concepts from a established theoretical basis (and also contribute to evolve such theoretical basis). When studies are not properly planned, they are not contributing to the theoretical basis of the area.

Considering these problems, we proposed in this PhD thesis the establishment of domain-specific mechanisms to help researchers to scope and design experimental studies on programming education. We chose the domain of the integration of software testing into introductory programming courses to explore as our domain of interest (JONES, 2001; BARBOSA *et al.*, 2003; EDWARDS, 2004; JANZEN; SAIEDIAN, 2008; WHALLEY; PHILPOTT, 2011). To this end, we created a framework for experimental studies on the integration of software testing into programming education, which is discussed throughout this thesis.

8.1 Contributions

We highlight the following contributions of this PhD work:

- **Identification of knowledge gaps caused by software testing education:** we conducted a survey to identify the knowledge gaps in software testing topics presented by graduates from Brazilian computing undergraduate programs (with respect to industry needs). To this end, we compared the software testing education delivered to them and the testing practices they have applied in industry. The results indicate which testing topics have been underemphasized and overemphasized throughout computing courses and, moreover, a lack of practice activities for all testing topics.
- **Overview of research and teaching practices on software testing in programming courses:** We conducted a systematic mapping of the literature on software testing in programming courses. We selected 293 relevant papers, which allowed us to characterize how instructors/researchers have been integrating software testing in programming courses and how they have been designing experimental studies in this domain. Additionally, the map emphasizes topics with research gaps in this domain.
- **Conduction of experiments about students' test design:** We executed two experiments in this domain, which contribute with their results and also their design, since both present variables were not yet investigated in existing studies.
- **Establishment of a framework for experimental studies on the integration of software testing into programming education:** an explicit framework that represents the underlying structure of experiments from a given domain within the scope of programming education

8.2 Limitations

Our choices to conduct this PhD work raise the following limitations:

- **Domain of programming education:** We addressed a limited domain of programming education in our framework. Other practices and approaches integrated into programming courses (e.g. pair programming, visualization, peer review etc) may also have associated variables to programming education.

- **Variables of the experimental framework:** The variables included in the experimental framework are not an exhaustive theoretical model of this domain. We rather aimed to achieving a representative model, i.e. which are able to accommodate existing studies and support the designing of future ones.
- **Conducted experiments:** Our conducted experiments both involved a heterogeneous population (i.e. students taking different courses), which can introduce confounding factors.
- **Framework instantiation:** We only instantiated retroactively our own experiments into the framework. There is the need to investigate the use of the framework to support the designing of new studies, specially by other researchers not involved with the framework establishment.

8.3 Future work

We indicate the following directions for future work:

- **Conduction of more experiments:** The experimental framework can help to design new studies and also to replicate existing ones. The replication of experimental studies is crucial to advance the body of knowledge in a domain. A given hypotheses should be explored in several studies in order to increase the confidence its acceptance. The framework can also help in defining the variations in the replication design.
- **Evolution of the framework:** Since it is not an exhaustive model, the framework can evolve using new sources of information, such as future studies in the same domain, studies investigating other teaching approaches in programming education, or even studies focused in industry testing practices, specially the ones adopted by developers (e.g. TDD).
- **Address research gaps identified in the systematic mapping:** Considering the distribution of papers of papers in the map, it is possible to note some research gaps, such as on the topic of course materials. In particular, considering the deficiencies in testing education pointed out by respondents on the survey, having more materials of testing leveled to novice programmers could help to address this issue and reduce knowledge gaps.
- **Meta-analysis of existing studies:** In order to conduct meta-analysis of a set of experiments, their results should be integrated. The framework can support this integration around the same variables and metrics.

8.4 Publications

The publications that resulted from this PhD work so far are the following:

- Scatalon, L.P.; Barbosa, E.F. and Garcia, R.E. **Challenges to integrate software testing into introductory programming courses**. In *47th Annual Frontiers in Education Conference (FIE 2017)*, Indianapolis, Indiana, USA, 2017.
- Scatalon, L.P.; Prates, J.M.; Souza, D.M.; Barbosa, E.F. and Garcia, R.E. **Towards the role of test design in programming assignments**. In *30th IEEE Conference on Software Engineering Education and Training (CSEE&T 2017)*, Savannah, Georgia, USA, 2017.
- Scatalon, L.P.; Fioravanti, M.L.; Prates, J.M.; Garcia, R.E and Barbosa, E.F. **A survey on graduates' curriculum-based knowledge gaps in software testing**. In *48th Annual Frontiers in Education Conference (FIE 2018)*, San Jose, California, USA.
- Scatalon, L.P.; Carver, J.C.; Garcia, R.E and Barbosa, E.F. **Software testing in introductory programming courses: A systematic mapping study**. In *50th ACM Technical Symposium on Computing Science Education (SIGCSE'19)*, Minneapolis, Minnesota, USA.

BIBLIOGRAPHY

AALTONEN, K.; IHANTOLA, P.; SEPPALA, O. 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**. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 153–160. ISBN 978-1-4503-0240-1. Available: <http://doi.acm.org/10.1145/1869542.1869567>. Citations on pages 92, 93, and 199.

ACM/IEEE-CS. **Computing Curricula 2001**. 2001. Joint Task Force on Computing Curricula, available at <www.acm.org/education/curric_vols/cc2001.pdf>. Citations on pages 13, 25, 26, 28, and 135.

____. **Computer Science Curricula 2013**. 2013. Joint Task Force on Computing Curricula, available at <www.acm.org/education/CS2013-final-report.pdf>. Citations on pages 13, 19, 25, 26, 27, 29, 30, and 35.

ADAMS, J. Test-driven Data Structures: Revitalizing CS2. In: **Proceedings of the 40th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2009. (SIGCSE '09), p. 143–147. ISBN 978-1-60558-183-5. Available: <http://doi.acm.org/10.1145/1508865.1508920>. Citation on page 191.

AGARWAL, R.; EDWARDS, S. H.; PEREZ-QUINONES, M. A. Designing an adaptive learning module to teach software testing. In: **Proceedings of the 37th SIGCSE technical symposium on Computer science education**. New York, NY, USA: ACM, 2006. (SIGCSE '06), p. 259–263. ISBN 1-59593-259-3. Available: <http://doi.acm.org/10.1145/1121341.1121420>. Citations on pages 94, 109, 128, and 193.

AHADI, A.; LISTER, R. Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of the one elephant? In: **Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)**. New York, NY, USA: ACM, 2013. p. 123–128. ISBN 978-1-4503-2243-0. Citation on page 31.

AKOUR, M. Towards Harnessing Testing Tools into Programming Courses Curricula: Case Study of Jordan. In: **2014 International Conference on Computational Science and Computational Intelligence**. [S.l.: s.n.], 2014. v. 2, p. 197–200. Citation on page 198.

AL-ZUBIDY, A.; CARVER, J. C.; HECKMAN, S.; SHERRIFF, M. A (updated) review of empiricism at the sigcse technical symposium. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)**. New York, NY, USA: ACM, 2016. p. 120–125. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844601>. Citations on pages 21, 78, and 123.

ALA-MUTKA, K. M. A survey of automated assessment approaches for programming assignments. **Computer Science Education**, Routledge, v. 15, n. 2, p. 83–102, 2005. Available: <https://doi.org/10.1080/08993400500150747>. Citation on page 197.

ALKADI, I.; ALKADI, G. To C++ or to Java, that is the question! Featuring a test plan and an automated testing assistant for object oriented testing. In: **Proceedings, IEEE Aerospace Conference**. [S.l.: s.n.], 2002. v. 7, p. 3679–3688. Citation on page 193.

ALLEN, E.; CARTWRIGHT, R.; REIS, C. Production Programming in the Classroom. In: **Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2003. (SIGCSE ’03), p. 89–93. ISBN 1-58113-648-X. Available: <http://doi.acm.org/10.1145/611892.611940>. Citation on page 194.

ALLEN, E.; CARTWRIGHT, R.; STOLER, B. DrJava: A Lightweight Pedagogic Environment for Java. In: **Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2002. (SIGCSE ’02), p. 137–141. ISBN 1-58113-473-8. Available: <http://doi.acm.org/10.1145/563340.563395>. Citation on page 197.

ALLEVATO, A.; EDWARDS, S. Dereferee: Instrumenting C++ pointers with meaningful runtime diagnostics. **Software - Practice and Experience**, v. 44, n. 8, p. 973–997, 2014. ISSN 00380644. Citations on pages 87, 99, 130, and 196.

ALLEVATO, A.; EDWARDS, S. H. RoboLIFT: Engaging CS2 Students with Testable, Automatically Evaluated Android Applications. In: **Proceedings of the 43rd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2012. (SIGCSE ’12), p. 547–552. ISBN 978-1-4503-1098-7. Available: <http://doi.acm.org/10.1145/2157136.2157293>. Citation on page 196.

ALLEVATO, A.; EDWARDS, S. H.; PEREZ-QUINONES, M. A. Dereferee: Exploring Pointer Mismanagement in Student Code. In: **Proceedings of the 40th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2009. (SIGCSE ’09), p. 173–177. ISBN 978-1-60558-183-5. Available: <http://doi.acm.org/10.1145/1508865.1508928>. Citations on pages 99 and 196.

ALLEVATO, A.; THORNTON, M.; EDWARDS, S. H.; PEREZ-QUINONES, M. A. Mining data from an automated grading and testing system by adding rich reporting capabilities. In: **Educational Data Mining (EDM 2008)**. [S.l.: s.n.], 2008. Citation on page 197.

ALLISON, C. D. The Simplest Unit Test Tool That Could Possibly Work. **J. Comput. Sci. Coll.**, v. 23, n. 1, p. 183–189, Oct. 2007. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1289280.1289318>. Citation on page 193.

ALLOWATT, A.; EDWARDS, S. H. Ide support for test-driven development and automated grading in both java and c++. In: **Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange**. New York, NY, USA: ACM, 2005. (eclipse ’05), p. 100–104. ISBN 1-59593-342-5. Available: <http://doi.acm.org/10.1145/1117696.1117717>. Citation on page 197.

ALMSTRUM, V. L.; HAZZAN, O.; GUZDIAL, M.; PETRE, M. Challenges to computer science education research. In: **Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2005. (SIGCSE ’05), p. 191–192. ISBN 1-58113-997-7. Available: <http://doi.acm.org/10.1145/1047344.1047415>. Citation on page 23.

AMELUNG, M.; FORBRIG, P.; RöSNER, D. Towards generic and flexible web services for e-assessment. In: **Proceedings of the 13th Annual Conference on Innovation and Technology**

in Computer Science Education. New York, NY, USA: ACM, 2008. (ITiCSE '08), p. 219–224. ISBN 978-1-60558-078-4. Available: <http://doi.acm.org/10.1145/1384271.1384330>. Citation on page 195.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 2. ed. New York, NY, USA: Cambridge University Press, 2016. ISBN 9781107172012. Citations on pages 104, 118, 119, and 124.

ANDRIANOFF, S. K.; LEVINE, D. B.; GEWAND, S. D.; HEISSENBERGER, G. A. A Testing-based Framework for Programming Contests. In: **Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange**. New York, NY, USA: ACM, 2003. (eclipse '03), p. 94–98. Available: <http://doi.acm.org/10.1145/965660.965680>. Citation on page 197.

BALDWIN, J.; CRUPI, E.; ESTRELLADO, T. Webwork for programming fundamentals. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 38, n. 3, p. 361–361, Jun. 2006. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1140123.1140272>. Citation on page 195.

BANDARA, R. **A Simple String Compression Algorithm**. 2011. Available at <https://www.codeproject.com/Articles/223610/A-Simple-String-Compression-Algorithm>. Citation on page 119.

BARBOSA, E. F.; MALDONADO, J. C.; LEBLANC, R.; GUZDIAL, M. Introducing testing practices into objects and design course. In: **Proceedings 16th Conference on Software Engineering Education and Training, 2003. (CSEE&T 2003).** [S.l.: s.n.], 2003. p. 279–286. ISSN 1093-0175. Citations on pages 20, 36, 109, 140, and 193.

BARBOSA, E. F.; SILVA, M. A. G.; CORTE, C. K. D.; MALDONADO, J. C. Integrated teaching of programming foundations and software testing. In: **2008 38th Annual Frontiers in Education Conference**. [S.l.: s.n.], 2008. p. S1H–5–S1H–10. Citations on pages 36, 74, 128, and 193.

BARR, V.; STEPHENSON, C. Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community? **ACM Inroads**, ACM, New York, NY, USA, v. 2, n. 1, p. 48–54, Feb. 2011. ISSN 2153-2184. Available: <http://doi.acm.org/10.1145/1929887.1929905>. Citation on page 19.

BARRIOCANAL, E. G.; URBAN, M.-A. S.; CUEVAS, I. A.; PEREZ, P. D. An Experience in Integrating Automated Unit Testing Practices in an Introductory Programming Course. **SIGCSE Bull.**, v. 34, n. 4, p. 125–128, Dec. 2002. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/820127.820183>. Citations on pages 36, 94, 99, and 192.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. Goal question metric paradigm. In: **Encyclopedia of Software Engineering**. [S.l.]: John Wiley & Sons, 1994. p. 528–532. Citations on pages 11, 44, 47, and 49.

BASILI, V. R.; GREEN, S.; LAITENBERGER, O.; LANUBILE, F.; SHULL, F.; SORUMGARD, S.; ZELKOWITZ, M. V. The empirical investigation of perspective-based reading. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 1, n. 2, p. 133–164, 1996. ISSN 1382-3256. Citations on pages 40 and 57.

BASILI, V. R.; SHULL, F.; LANUBILE, F. Building knowledge through families of experiments. **IEEE Transactions on Software Engineering**, v. 25, n. 4, p. 456–473, 1999. Citations on pages 11, 13, 23, 39, 55, 56, 57, 60, 62, 124, and 125.

BASILI, V. R.; ZELKOWITZ, M. V. Empirical studies to build a science of computer science. **Communications of the ACM**, ACM, New York, NY, USA, v. 50, n. 11, p. 33–37, 2007. Citation on page 40.

____. Empirical studies to build a science of computer science. **Commun. ACM**, ACM, New York, NY, USA, v. 50, n. 11, p. 33–37, Nov. 2007. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/1297797.1297819>. Citation on page 55.

BASU, S.; WU, A.; HOU, B.; DENERO, J. Problems before solutions: Automated problem clarification at scale. In: **Proceedings of the Second (2015) ACM Conference on Learning Scale**. New York, NY, USA: ACM, 2015. (L@S '15), p. 205–213. ISBN 978-1-4503-3411-2. Available: <http://doi.acm.org/10.1145/2724660.2724679>. Citation on page 193.

BAUMSTARK JR., L.; ORSEGA, M. Quantifying Introductory CS Students' Iterative Software Process by Mining Version Control System Repositories. **J. Comput. Sci. Coll.**, v. 31, n. 6, p. 97–104, Jun. 2016. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2904446.2904470>. Citations on pages 100, 103, 135, and 200.

BEAUBOUEF, T.; MASON, J. Why the high attrition rate for computer science students: Some thoughts and observations. **SIGCSE Bulletin**, ACM, New York, NY, USA, v. 37, n. 2, p. 103–106, Jun. 2005. ISSN 0097-8418. Citations on pages 20 and 31.

BEAUBOUEF, T.; ZHANG, W. Learning to Program Through Use of Code Verification. **J. Comput. Sci. Coll.**, v. 27, n. 5, p. 78–84, May 2012. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2168874.2168895>. Citation on page 193.

BECK, L. L.; CHIZHIK, A. W.; MCELROY, A. C. Cooperative learning techniques in cs1: Design and experimental evaluation. In: **Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)**. New York, NY, USA: ACM, 2005. p. 470–474. ISBN 1-58113-997-7. Citation on page 35.

BECKER, B. W. Teaching cs1 with karel the robot in java. In: **Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)**. New York, NY, USA: ACM, 2001. p. 50–54. ISBN 1-58113-329-4. Citation on page 38.

BELL, J.; SHETH, S.; KAISER, G. Secret Ninja Testing with HALO Software Engineering. In: **Proceedings of the 4th International Workshop on Social Software Engineering**. New York, NY, USA: ACM, 2011. (SSE '11), p. 43–47. ISBN 978-1-4503-0850-2. Available: <http://doi.acm.org/10.1145/2024645.2024657>. Citation on page 198.

BEN-ARI, M.; BERGLUND, A.; BOOTH, S.; HOLMBOE, C. What do we mean by theoretically sound research in computer science education? In: **Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2004. (ITiCSE '04), p. 230–231. ISBN 1-58113-836-9. Available: <http://doi.acm.org/10.1145/1007996.1008059>. Citations on pages 21 and 139.

BENNEDSEN, J.; CASPERSEN, M. E. Revealing the Programming Process. In: **Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2005. (SIGCSE '05), p. 186–190. ISBN 1-58113-997-7. Available: <http://doi.acm.org/10.1145/1047344.1047413>. Citation on page 195.

____. Failure rates in introductory programming. **SIGCSE Bulletin**, ACM, New York, NY, USA, v. 39, n. 2, p. 32–36, Jun. 2007. ISSN 0097-8418. Citation on page 20.

BERGLUND, A.; DANIELS, M.; PEARS, A. Qualitative research projects in computing education research: An overview. In: **Proceedings of the 8th Australasian Conference on Computing Education (ACE '06)**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006. p. 25–33. ISBN 1-920682-34-1. Citations on pages 21 and 139.

BIRCH, G.; FISCHER, B.; POPPLETON, M. Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment. In: **Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2016. (ITiCSE '16), p. 168–173. ISBN 978-1-4503-4231-5. Available: <http://doi.acm.org/10.1145/2899415.2899433>. Citation on page 197.

BISHOP, J.; HORSPOOL, R. N.; XIE, T.; TILLMANN, N.; HALLEUX, J. de. Code Hunt: Experience with Coding Contests at Scale. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 2**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 398–407. Available: <http://dl.acm.org/citation.cfm?id=2819009.2819072>. Citation on page 196.

BLAHETA, D. Unci: A C++-based Unit-testing Framework for Intro Students. In: **Proceedings of the 46th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 475–480. ISBN 978-1-4503-2966-8. Available: <http://doi.acm.org/10.1145/2676723.2677228>. Citations on pages 87, 91, 130, and 196.

BRADSHAW, M. K. Ante Up: A Framework to Strengthen Student-Based Testing of Assignments. In: **Proceedings of the 46th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 488–493. ISBN 978-1-4503-2966-8. Available: <http://doi.acm.org/10.1145/2676723.2677247>. Citation on page 196.

BRANNOCK, E.; NAPIER, N. Real-world Testing: Using FOSS for Software Development Courses. In: **Proceedings of the 13th Annual Conference on Information Technology Education**. New York, NY, USA: ACM, 2012. (SIGITE '12), p. 87–88. ISBN 978-1-4503-1464-0. Available: <http://doi.acm.org/10.1145/2380552.2380578>. Citation on page 193.

BRAUGHT, G.; MIDKIFF, J. Tool Design and Student Testing Behavior in an Introductory Java Course. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 449–454. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844641>. Citations on pages 87, 88, 90, 91, 92, 94, 97, 130, 132, and 197.

BRERETON, P.; KITCHENHAM, B. A.; BUDGEN, D.; TURNER, M.; KHALIL, M. Lessons from applying the systematic literature review process within the software engineering domain. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 80, n. 4, p. 571–583, Apr. 2007. ISSN 0164-1212. Citation on page 23.

BRIAN, S. A.; THOMAS, R. N.; HOGAN, J. M.; FIDGE, C. Planting Bugs: A System for Testing Students' Unit Tests. In: **Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2015. (ITiCSE '15), p. 45–50. ISBN 978-1-4503-3440-2. Available: <http://doi.acm.org/10.1145/2729094.2742631>. Citation on page 198.

BRIGGS, T.; GIRARD, C. D. Tools and Techniques for Test-driven Learning in CS1. **J. Comput. Sci. Coll.**, v. 22, n. 3, p. 37–43, Jan. 2007. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1181849.1181854>. Citation on page 193.

BRITO, M. A. S.; ROSSI, J. L.; SOUZA, S. R. S. de; BRAGA, R. T. V. An Experience on Applying Software Testing for Teaching Introductory Programming Courses. **CLEI Electronic Journal**, scielouy, v. 15, p. 5 – 5, 04 2012. ISSN 0717-5000. Available: <http://www.scielo.edu.uy/scielo.php?script=sci_arttext&pid=S0717-50002012000100005&nrm=iso>. Citations on pages 87, 88, 89, 101, and 199.

BROWN, C.; PASTEL, R.; SIEVER, B.; EARNEST, J. JUG: A JUnit Generation, Time Complexity Analysis and Reporting Tool to Streamline Grading. In: **Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2012. (ITiCSE '12), p. 99–104. ISBN 978-1-4503-1246-2. Available: <http://doi.acm.org/10.1145/2325296.2325323>. Citation on page 198.

BRUCE, K. B. Controversy on how to teach cs 1: A discussion on the sigcse-members mailing list. **SIGCSE Bulletin**, ACM, New York, NY, USA, v. 37, n. 2, p. 111–117, Jun. 2005. ISSN 0097-8418. Citation on page 28.

BRUCE, K. B.; DANYLUK, A.; MURTAGH, T. Introducing concurrency in cs 1. In: **Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)**. New York, NY, USA: ACM, 2010. p. 224–228. ISBN 978-1-4503-0006-3. Citation on page 30.

BRYCE, R. Bug Wars: A Competitive Exercise to Find Bugs in Code. **J. Comput. Sci. Coll.**, v. 27, n. 2, p. 43–50, Dec. 2011. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2038836.2038842>. Citation on page 194.

BUDGEN, D.; BAILEY, J.; TURNER, M.; KITCHENHAM, B.; BRERETON, P.; CHARTERS, S. Cross-domain investigation of empirical practices. **IET Software**, v. 3, n. 5, p. 410–421, 2009. Citations on pages 20 and 23.

BUFFARDI, K.; EDWARDS, S. H. 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**. New York, NY, USA: ACM, 2012. (ITiCSE '12), p. 105–110. ISBN 978-1-4503-1246-2. Available: <http://doi.acm.org/10.1145/2325296.2325324>. Citations on pages 88, 90, 91, 92, 93, 97, and 200.

____. Impacts of teaching test-driven development to novice programmers. **International Journal of Information and Computer Science**, v. 1, n. 6, p. 135–143, 2012. ISSN 2161-6450. Available: <http://www.seipub.org/ijics/paperInfo.aspx?ID=1847>. Citations on pages 97, 132, and 194.

____. Effective and Ineffective Software Testing Behaviors by Novice Programmers. In: **Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research**. New York, NY, USA: ACM, 2013. (ICER '13), p. 83–90. ISBN 978-1-4503-2243-0. Available: <http://doi.acm.org/10.1145/2493394.2493406>. Citations on pages 90, 92, 95, 97, 101, 132, 134, and 200.

____. Impacts of Adaptive Feedback on Teaching Test-driven Development. In: **Proceeding of the 44th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2013. (SIGCSE '13), p. 293–298. ISBN 978-1-4503-1868-6. Available: <http://doi.acm.org/10.1145/2445196.2445287>. Citations on pages 87, 95, 97, 101, 130, 132, and 196.

____. A Formative Study of Influences on Student Testing Behaviors. In: **Proceedings of the 45th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 597–602. ISBN 978-1-4503-2605-6. Available: <http://doi.acm.org/10.1145/2538862.2538982>. Citations on pages 87, 94, 100, 102, 130, 132, 134, and 200.

____. Responses to Adaptive Feedback for Software Testing. In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education**. New York, NY, USA: ACM, 2014. (ITiCSE '14), p. 165–170. ISBN 978-1-4503-2833-3. Available: <http://doi.acm.org/10.1145/2591708.2591756>. Citations on pages 87, 99, 130, and 196.

____. Reconsidering Automated Feedback: A Test-Driven Approach. In: **Proceedings of the 46th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 416–420. ISBN 978-1-4503-2966-8. Available: <http://doi.acm.org/10.1145/2676723.2677313>. Citations on pages 99 and 196.

CAMARA, B. H. P.; SILVA, M. A. G. A Strategy to Combine Test-Driven Development and Test Criteria to Improve Learning of Programming Skills. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 443–448. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844633>. Citations on pages 87, 91, 92, 101, 129, and 194.

CARBONE, A.; HURST, J.; MITCHELL, I.; GUNSTONE, D. Principles for designing programming exercises to minimise poor learning behaviours in students. In: **Proceedings of the Australasian Conference on Computing Education**. New York, NY, USA: ACM, 2000. (ACSE '00), p. 26–33. ISBN 1-58113-271-9. Available: <http://doi.acm.org/10.1145/359369.359374>. Citation on page 194.

CARDELL-OLIVER, R.; ZHANG, L.; BARADY, R.; LIM, Y. H.; NAVEED, A.; WOODINGS, T. Automated Feedback for Quality Assurance in Software Engineering Education. In: **2010 21st Australian Software Engineering Conference**. [S.l.: s.n.], 2010. p. 157–164. Citations on pages 88, 89, 90, 92, and 196.

CARLSON, B. An Agile classroom experience: Teaching TDD and refactoring. In: **Proceedings - Agile 2008 Conference**. [S.l.: s.n.], 2008. p. 465–469. ISBN 978-0-7695-3321-6. Citation on page 192.

CARTER, J.; WHITE, S.; FRASER, K.; KURKOVSKY, S.; MCCREESH, C.; WIECK, M. Iticse 2010 working group report: Motivating our top students. In: **Proceedings of the 2010 ITiCSE Working Group Reports (ITiCSE-WGR '10)**. New York, NY, USA: ACM, 2010. p. 29–47. ISBN 978-1-4503-0677-5. Citation on page 117.

CARVER, J.; JACCHERI, L.; MORASCA, S.; SHULL, F. Issues in using students in empirical studies in software engineering education. In: **Software Metrics Symposium**. [S.l.: s.n.], 2003. p. 239–249. ISSN 1530-1435. Citation on page 45.

CARVER, J. C.; JACCHERI, L.; MORASCA, S.; SHULL, F. A checklist for integrating student empirical studies with research and teaching goals. **Empirical Software Engineering**, Springer US, v. 15, n. 1, p. 35–59, 2010. ISSN 1382-3256. Citation on page 45.

CARVER, J. C.; KRAFT, N. A. Evaluating the testing ability of senior-level computer science students. In: **24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)**. [S.l.: s.n.], 2011. p. 169–178. Citations on pages 63, 75, 117, 120, and 121.

CASPERSEN, M. E.; KOLLING, M. A Novice's Process of Object-oriented Programming. In: **Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications**. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 892–900. ISBN 1-59593-491-X. Available: <http://doi.acm.org/10.1145/1176617.1176741>. Citation on page 195.

____. STREAM: A First Programming Process. **Trans. Comput. Educ.**, v. 9, n. 1, p. 4:1–4:29, Mar. 2009. ISSN 1946-6226. Available: <http://doi.acm.org/10.1145/1513593.1513597>. Citation on page 194.

CHEANG, B.; KURNIA, A.; LIM, A.; OON, W.-C. On automated grading of programming assignments in an academic institution. **Comput. Educ.**, Elsevier Science Ltd., Oxford, UK, UK, v. 41, n. 2, p. 121–131, Sep. 2003. ISSN 0360-1315. Available: <http://dx.doi.org/10.1016/S0360-1315(03)00030-7>. Citation on page 195.

CHEN, W. K.; HALL, B. R. Applying Software Engineering in CS1. In: **Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2013. (ITiCSE '13), p. 297–302. ISBN 978-1-4503-2078-8. Available: <http://doi.acm.org/10.1145/2462476.2462480>. Citation on page 193.

CHOY, M.; NAZIR, U.; POON, C. K.; YU, Y. T. Experiences in using an automated system for improving students' learning of computer programming. In: LAU, R. W. H.; LI, Q.; CHEUNG, R.; LIU, W. (Ed.). **Advances in Web-Based Learning – ICWL 2005**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 267–272. ISBN 978-3-540-31716-6. Available: <https://doi.org/10.1007/11528043_26>. Citation on page 195.

CHRISTENSEN, H. B. 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**. New York, NY, USA: ACM, 2003. (ITiCSE '03), p. 7–10. ISBN 1-58113-672-2. Available: <http://doi.acm.org/10.1145/961511.961517>. Citations on pages 36 and 191.

CLARKE, P. J.; ALLEN, A. A.; KING, T. M.; JONES, E. L.; NATESAN, P. Using a Web-based Repository to Integrate Testing Tools into Programming Courses. In: **Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion**. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 193–200. ISBN 978-1-4503-0240-1. Available: <http://doi.acm.org/10.1145/1869542.1869573>. Citation on page 196.

CLEGG, B. S.; ROJAS, J. M.; FRASER, G. Teaching software testing concepts using a mutation testing game. In: **Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track**. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE-SEET '17), p. 33–36. ISBN 978-1-5386-2671-9. Available: <https://doi.org/10.1109/ICSE-SEET.2017.1>. Citation on page 198.

CLEMENTS, J.; JANZEN, D. Overcoming Obstacles to Test-Driven Learning on Day One. In: **2010 Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.: s.n.], 2010. p. 448–453. Citation on page 197.

COLLOFELLO, J.; VEHATHIRI, K. An environment for training computer science students on software testing. In: **Proceedings Frontiers in Education 35th Annual Conference**. [S.l.: s.n.], 2005. p. T3E–6. ISSN 0190-5848. Citations on pages 109 and 195.

COMBEFIS, S.; PAQUES, A. Pythia Reloaded: An Intelligent Unit Testing-based Code Grader for Education. In: **Proceedings of the 1st International Workshop on Code Hunt Workshop on Educational Software Engineering**. New York, NY, USA: ACM, 2015. (CHESE 2015), p. 5–8. ISBN 978-1-4503-3711-3. Available: <http://doi.acm.org/10.1145/2792404.2792407>. Citation on page 198.

COWDEN, D.; O'NEILL, A.; OPAVSKY, E.; USTEK, D.; WALKER, H. M. A c-based introductory course using robots. In: **Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE '12)**. New York, NY, USA: ACM, 2012. p. 27–32. ISBN 978-1-4503-1098-7. Citations on pages 30 and 135.

COWLING, T. Stages in teaching software testing. In: **2012 34th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2012. p. 1185–1194. Citations on pages 36 and 191.

DALY, C.; HORGAN, J. M. An automated learning system for java programming. **IEEE Transactions on Education**, v. 47, n. 1, p. 10–17, Feb 2004. ISSN 0018-9359. Citations on pages 87, 94, 95, 98, 99, 130, and 196.

DANUTAMA, K.; LIEM, I. Scalable Autograder and LMS Integration. **Procedia Technology**, v. 11, p. 388 – 395, 2013. ISSN 2212-0173. Available: <http://www.sciencedirect.com/science/article/pii/S2212017313003617>. Citation on page 198.

DAVIES, S.; POLACK-WAHL, J. A.; ANEWALT, K. 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)**. New York, NY, USA: ACM, 2011. p. 625–630. ISBN 978-1-4503-0500-6. Citations on pages 30 and 135.

DEKHANE, S.; PRICE, R. Course-embedded research in software development courses. In: **45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)**. New York, NY, USA: ACM, 2014. p. 45–48. ISBN 978-1-4503-2605-6. Citation on page 45.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introducao ao teste de software *(Introduction to software testing)***. 2nd edition. ed. [S.l.]: Elsevier, 2016. Citation on page 65.

DELIGIANNIS, I. S.; SHEPPERD, M.; WEBSTER, S.; ROUMELIOTIS, M. A review of experimental investigations into object-oriented technology. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 7, n. 3, p. 193–231, 2002. ISSN 1382-3256. Citation on page 40.

DENNY, P.; LUXTON-REILLY, A.; TEMPERO, E.; HENDRICKX, J. CodeWrite: Supporting Student-driven Practice of Java. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 471–476. ISBN 978-1-4503-0500-6. Available: <http://doi.acm.org/10.1145/1953163.1953299>. Citations on pages 88, 89, 98, 99, 101, and 196.

DESAI, C.; JANZEN, D.; SAVAGE, K. A survey of evidence for test-driven development in academia. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 40, n. 2, p. 97–101, Jun. 2008. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1383602.1383644>. Citation on page 194.

DESAI, C.; JANZEN, D. S.; CLEMENTS, J. Implications of Integrating Test-driven Development into CS1/CS2 Curricula. In: **Proceedings of the 40th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2009. (SIGCSE '09), p. 148–152.

ISBN 978-1-60558-183-5. Available: <http://doi.acm.org/10.1145/1508865.1508921>. Citations on pages 36, 128, and 193.

DEWEY, K.; CONRAD, P.; CRAIG, M.; MOROZOVA, E. Evaluating test suite effectiveness and assessing student code via constraint logic programming. In: **Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2017. (ITiCSE '17), p. 317–322. ISBN 978-1-4503-4704-4. Available: <http://doi.acm.org/10.1145/3059009.3059051>. Citation on page 198.

DORIN, P. M. Laboratory Redux. **SIGCSE Bull.**, v. 39, n. 2, p. 84–87, Jun. 2007. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1272848.1272891>. Citation on page 191.

DOUCE, C.; LIVINGSTONE, D.; ORWELL, J. Automatic Test-based Assessment of Programming: A Review. **J. Educ. Resour. Comput.**, v. 5, n. 3, Sep. 2005. ISSN 1531-4278. Available: <http://doi.acm.org/10.1145/1163405.1163409>. Citation on page 197.

DVORNIK, T.; JANZEN, D. S.; CLEMENTS, J.; DEKHTYAR, O. Supporting introductory test-driven labs with WebIDE. In: **2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T)**. [S.l.: s.n.], 2011. p. 51–60. Citations on pages 87, 94, 101, 130, and 196.

DYBA, T.; DINGSOYR, T. Empirical studies of agile software development: A systematic review. **Information and Software Technology**, v. 50, n. 9–10, p. 833–859, 2008. ISSN 0950-5849. Citation on page 40.

EARLE, C. B.; FREDLUND, L.-\.; HUGHES, J. Automatic Grading of Programming Exercises Using Property-Based Testing. In: **Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2016. (ITiCSE '16), p. 47–52. ISBN 978-1-4503-4231-5. Available: <http://doi.acm.org/10.1145/2899415.2899443>. Citation on page 197.

EASTERBROOK, S.; SINGER, J.; STOREY, M.-A.; DAMIAN, D. Selecting empirical methods for software engineering research. In: SHULL, F.; SINGER, J.; SJOBERG, D. I. (Ed.). **Guide to Advanced Empirical Software Engineering**. [S.l.]: Springer London, 2008. p. 285–311. ISBN 978-1-84800-043-8. Citations on pages 21, 40, and 56.

EBEL, G.; BEN-ARI, M. Affective effects of program visualization. In: **Proceedings of the Second International Workshop on Computing Education Research (ICER '06)**. New York, NY, USA: ACM, 2006. p. 1–5. ISBN 1-59593-494-4. Citation on page 37.

EDWARDS, S.; LI, Z. Towards Progress Indicators for Measuring Student Programming Effort During Solution Development. In: **Proceedings of the 16th Koli Calling International Conference on Computing Education Research**. New York, NY, USA: ACM, 2016. (Koli Calling '16), p. 31–40. ISBN 978-1-4503-4770-9. Available: <http://doi.acm.org/10.1145/2999541.2999561>. Citations on pages 100, 133, and 199.

EDWARDS, S. H. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. **J. Educ. Resour. Comput.**, v. 3, n. 3, Sep. 2003. ISSN 1531-4278. Available: <http://doi.acm.org/10.1145/1029994.1029995>. Citations on pages 86, 89, 90, 91, 92, 93, 97, 98, and 192.

____. 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**. New York, NY, USA: ACM, 2003. (OOPSLA '03), p. 148–155. ISBN 1-58113-751-6. Available: <http://doi.acm.org/10.1145/949344.949390>. Citations on pages 75 and 191.

____. Teaching Software Testing: Automatic Grading Meets Test-first Coding. In: **Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications**. New York, NY, USA: ACM, 2003. (OOPSLA '03), p. 318–319. ISBN 1-58113-751-6. Available: <http://doi.acm.org/10.1145/949344.949431>. Citation on page 192.

____. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In: **International Conference on Education and Information Systems: Technologies and Applications (EISTA?03)**. [s.n.], 2003. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.138.3615>. Citations on pages 87, 90, 91, 92, 93, 97, and 194.

____. 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**. New York, NY, USA: ACM, 2004. (SIGCSE '04), p. 26–30. ISBN 1-58113-798-2. Available: <http://doi.acm.org/10.1145/971300.971312>. Citations on pages 20, 35, 75, 86, 90, 91, 92, 93, 97, 140, and 192.

____. Work-in-progress: Program Grading and Feedback Generation with Web-CAT. In: **Proceedings of the First ACM Conference on Learning @ Scale Conference**. New York, NY, USA: ACM, 2014. (L@S '14), p. 215–216. ISBN 978-1-4503-2669-8. Available: <http://doi.acm.org/10.1145/2556325.2567888>. Citation on page 196.

EDWARDS, S. H.; ALLEVATO, A. 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)**. New York, NY, USA: ACM, 2013. p. 321–321. ISBN 978-1-4503-2078-8. Citations on pages 30 and 135.

EDWARDS, S. H.; BöRSTLER, J.; CASSEL, L. N.; HALL, M. S.; HOLLINGSWORTH, J. Developing a common format for sharing programming assignments. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 40, n. 4, p. 167–182, Nov. 2008. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1473195.1473240>. Citation on page 194.

EDWARDS, S. H.; PEREZ-QUINONES, M. A. Experiences Using Test-driven Development with an Automated Grader. **J. Comput. Sci. Coll.**, v. 22, n. 3, p. 44–50, Jan. 2007. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1181849.1181855>. Citation on page 192.

____. Web-CAT: Automatically Grading Programming Assignments. In: **Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2008. (ITiCSE '08), p. 328–328. ISBN 978-1-60558-078-4. Available: <http://doi.acm.org/10.1145/1384271.1384371>. Citation on page 197.

EDWARDS, S. H.; SHAMS, Z. Comparing Test Quality Measures for Assessing Student-written Tests. In: **Companion Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE Companion 2014), p. 354–363. ISBN

978-1-4503-2768-8. Available: <http://doi.acm.org/10.1145/2591062.2591164>. Citations on pages 90, 92, 93, 94, 132, and 199.

____. Do Student Programmers All Tend to Write the Same Software Tests? In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education**. New York, NY, USA: ACM, 2014. (ITiCSE '14), p. 171–176. ISBN 978-1-4503-2833-3. Available: <http://doi.acm.org/10.1145/2591708.2591757>. Citations on pages 91, 93, 94, 109, 117, 131, and 200.

EDWARDS, S. H.; SHAMS, Z.; COGSWELL, M.; SENKBEIL, R. C. 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**. New York, NY, USA: ACM, 2012. (SIGCSE '12), p. 221–226. ISBN 978-1-4503-1098-7. Available: <http://doi.acm.org/10.1145/2157136.2157202>. Citations on pages 90, 93, and 196.

EDWARDS, S. H.; SHAMS, Z.; ESTEP, C. Adaptively Identifying Non-terminating Code when Testing Student Programs. In: **Proceedings of the 45th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2014. (SIGCSE '14), p. 15–20. ISBN 978-1-4503-2605-6. Available: <http://doi.acm.org/10.1145/2538862.2538926>. Citations on pages 99, 100, and 196.

EDWARDS, S. H.; SNYDER, J.; nONES, M. A. P.-Q.; ALLEVATO, A.; KIM, D.; TRETOLA, B. Comparing effective and ineffective behaviors of student programmers. In: **Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09)**. New York, NY, USA: ACM, 2009. p. 3–14. ISBN 978-1-60558-615-1. Citation on page 117.

ELBAUM, S.; PERSON, S.; DOKULIL, J.; JORDE, M. Bug hunt: Making early software testing lessons engaging and affordable. In: **Proceedings - International Conference on Software Engineering**. [S.l.: s.n.], 2007. p. 687–697. ISBN 0-7695-2828-7 978-0-7695-2828-1. Citations on pages 103, 109, and 195.

ELOY, A. A. d. S.; MARTINS, A. R. Q.; PAZINATO, A. M.; LUKJANENKO, M. d. F. S. P.; LOPES, R. d. D. Programming literacy: Computational thinking in brazilian public schools. In: **Proceedings of the 2017 Conference on Interaction Design and Children**. New York, NY, USA: ACM, 2017. (IDC '17), p. 439–444. ISBN 978-1-4503-4921-5. Available: <http://doi.acm.org/10.1145/3078072.3084306>. Citation on page 19.

ENGSTRöM, E.; RUNESON, P. A qualitative survey of regression testing practices. In: **Proceedings of the 11th International Conference on Product-Focused Software Process Improvement (PROFES'10)**. Berlin, Heidelberg: Springer-Verlag, 2010. p. 3–16. ISBN 3-642-13791-1, 978-3-642-13791-4. Citation on page 64.

ENSTROM, E.; KREITZ, G.; NIEMELA, F.; SODERMAN, P.; KANN, V. Five years with kattis: Using an automated assessment system in teaching. In: **2011 Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2011. p. T3J–1–T3J–6. Citation on page 196.

ERDOGMUS, H.; MORISIO, M.; TORCHIANO, M. On the effectiveness of the test-first approach to programming. **IEEE Transactions on Software Engineering**, v. 31, n. 3, p. 226–237, March 2005. ISSN 0098-5589. Citations on pages 87, 89, 96, 104, 129, and 194.

ETTEBERG, H. T.; AALBERG, T. JExercise: A Specification-based and Test-driven Exercise Support Plugin for Eclipse. In: **Proceedings of the 2006 OOPSLA Workshop on Eclipse**

Technology eXchange. New York, NY, USA: ACM, 2006. (eclipse '06), p. 70–74. ISBN 1-59593-621-1. Available: <http://doi.acm.org/10.1145/1188835.1188850>. Citation on page 195.

FAGIN, B.; MERKLE, L. Measuring the effectiveness of robots in teaching computer science. In: **Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)**. New York, NY, USA: ACM, 2003. p. 307–311. ISBN 1-58113-648-X. Citation on page 38.

FELLEISEN, M.; FINDLER, R. B.; FLATT, M.; KRISHNAMURTHI, S. The teachscheme! project: Computing and programming for every student. **Computer Science Education**, v. 14, n. 1, p. 55–77, 2004. Citation on page 29.

FENG, M. Y.; MCALLISTER, A. A tool for automated gui program grading. In: **Proceedings. Frontiers in Education. 36th Annual Conference**. [S.l.: s.n.], 2006. p. 7–12. ISSN 0190-5848. Citation on page 197.

FIDGE, C.; HOGAN, J.; LISTER, R. What vs. How: Comparing Students' Testing and Coding Skills. In: **Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2013. (ACE '13), p. 97–106. ISBN 978-1-921770-21-0. Available: <http://dl.acm.org/citation.cfm?id=2667199.2667210>. Citations on pages 90, 92, 102, 103, 116, 134, 135, and 200.

FINCHER, S.; PETRE, M. **Computer Science Education Research**. [S.l.]: Routledge, 2004. ISBN 9026519699. Citations on pages 20, 21, 123, and 139.

FINCHER, S.; TENENBERG, J.; ROBINS, A. Research design: Necessary bricolage. In: **Proceedings of the Seventh International Workshop on Computing Education Research**. New York, NY, USA: ACM, 2011. (ICER '11), p. 27–32. ISBN 978-1-4503-0829-8. Available: <http://doi.acm.org/10.1145/2016911.2016919>. Citation on page 23.

FISCHER, G.; GUDENBERG, J. W. von. Improving the Quality of Programming Education by Online Assessment. In: **Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java**. New York, NY, USA: ACM, 2006. (PPPJ '06), p. 208–211. ISBN 3-939352-05-5. Available: <http://doi.acm.org/10.1145/1168054.1168085>. Citation on page 195.

FORTE, A.; GUZDIAL, M. Motivation and non-majors in computer science: identifying discrete audiences for introductory courses. **IEEE Transactions on Education**, v. 48, n. 2, p. 248–253, May 2005. ISSN 0018-9359. Citations on pages 19 and 37.

FREZZA, S. Integrating testing and design methods for undergraduates: teaching software testing in the context of software design. In: **32nd Annual Frontiers in Education**. [S.l.: s.n.], 2002. v. 3, p. S1G–1–S1G–4 vol.3. ISSN 0190-5848. Citations on pages 109 and 191.

FU, X.; PELTSVERGER, B.; QIAN, K.; TAO, L.; LIU, J. APOGEE: Automated Project Grading and Instant Feedback System for Web Based Computing. In: **Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2008. (SIGCSE '08), p. 77–81. ISBN 978-1-59593-799-5. Available: <http://doi.acm.org/10.1145/1352135.1352163>. Citation on page 197.

FUNABIKI, N.; KUSAKA, R.; ISHIHARA, N.; KAO, W. C. A Proposal of Test Code Generation Tool for Java Programming Learning Assistant System. In: **2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)**. [S.l.: s.n.], 2017. p. 51–56. Citation on page 198.

FUNABIKI, N.; NAKAMURA, T.; KAO, W.-C. A proposal of Javadoc hint function for Java Programming Learning Assistant System. In: **2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)**. [S.l.: s.n.], 2014. p. 304–308. Citation on page 198.

GALLIS, H.; ARISHOLM, E.; DYBA, T. A transition from partner programming to pair programming – an industrial case study. In: **Pair Programming Work Shop in 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)**. [S.l.: s.n.], 2002. Citation on page 59.

____. An initial framework for research on pair programming. In: **International Symposium on Empirical Software Engineering (ISESE 2003)**. [S.l.: s.n.], 2003. p. 132–142. Citations on pages 11, 13, 22, 23, 55, 57, 59, 60, 62, and 124.

GAO, J.; PANG, B.; LUMETTA, S. S. Automated Feedback Framework for Introductory Programming Courses. In: **Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2016. (ITiCSE '16), p. 53–58. ISBN 978-1-4503-4231-5. Available: <http://doi.acm.org/10.1145/2899415.2899440>. Citation on page 196.

GAROUSI, V.; ZHI, J. A survey of software testing practices in Canada. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 86, n. 5, p. 1354–1376, May 2013. ISSN 0164-1212. Citations on pages 64, 65, and 75.

GASPAR, A.; LANGEVIN, S. Active learning in introductory programming courses through student-led live coding and test-driven pair programming. In: **EISTA 2007, Education and Information Systems, Technologies and Applications**. [s.n.], 2007. Available: <https://www.semanticscholar.org/paper/Active-learning-in-introductory-programming-course-Gaspar-Langevin/8d6a1c7d90d5bdf50d1a23b25dac6550785e6392>. Citations on pages 135 and 193.

____. Restoring "Coding with Intention" in Introductory Programming Courses. In: **Proceedings of the 8th ACM SIGITE Conference on Information Technology Education**. New York, NY, USA: ACM, 2007. (SIGITE '07), p. 91–98. ISBN 978-1-59593-920-3. Available: <http://doi.acm.org/10.1145/1324302.1324323>. Citations on pages 135 and 193.

GASPAR, A.; LANGEVIN, S.; BOYER, N.; TINDELL, R. A Preliminary Review of Undergraduate Programming Students' Perspectives on Writing Tests, Working with Others, and Using Peer Testing. In: **Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education**. New York, NY, USA: ACM, 2013. (SIGITE '13), p. 109–114. ISBN 978-1-4503-2239-3. Available: <http://doi.acm.org/10.1145/2512276.2512301>. Citation on page 193.

GESTWICKI, P. Design and evaluation of an undergraduate course on software development practices. In: **Proceedings of the 49th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 221–226. ISBN 978-1-4503-5103-4. Available: <http://doi.acm.org/10.1145/3159450.3159542>. Citation on page 191.

GHAFARIAN, A. Incorporating a Semester-long Project into the CS 2 Course. **J. Comput. Sci. Coll.**, v. 17, n. 2, p. 183–190, Dec. 2001. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=775339.775373>. Citation on page 194.

GIRARD, C. D.; WELLINGTON, C. Work in progress: A test-first approach to teaching cs1. In: **Proceedings. Frontiers in Education. 36th Annual Conference**. [S.l.: s.n.], 2006. p. 19–20. ISSN 0190-5848. Citation on page 193.

GOLDWASSER, M. H. A Gimmick to Integrate Software Testing Throughout the Curriculum. In: **Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2002. (SIGCSE '02), p. 271–275. ISBN 1-58113-473-8. Available: <http://doi.acm.org/10.1145/563340.563446>. Citation on page 192.

GOMEZ, O. S.; JURISTO, N.; VEGAS, S. Replications types in experimental disciplines. In: **Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)**. New York, NY, USA: ACM, 2010. p. 3:1–3:10. ISBN 978-1-4503-0039-1. Citation on page 42.

GóMEZ, O. S.; VEGAS, S.; JURISTO, N. Impact of cs programs on the quality of test cases generation: An empirical study. In: **Proceedings of the 38th International Conference on Software Engineering Companion**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 374–383. ISBN 978-1-4503-4205-6. Available: <http://doi.acm.org/10.1145/2889160.2889190>. Citations on pages 86, 87, 94, 103, 128, 133, and 199.

GONZALEZ-GUERRA, L. H.; LEAL-FLORES, A. J. Tutoring model to guide students in programming courses to create complete and correct solutions. In: **2014 9th International Conference on Computer Science Education**. [S.l.: s.n.], 2014. p. 75–80. Citation on page 192.

GOTEL, O.; SCHARFF, C.; WILDENBERG, A. Extending and Contributing to an Open Source Web-based System for the Assessment of Programming Problems. In: **Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java**. New York, NY, USA: ACM, 2007. (PPPJ '07), p. 3–12. ISBN 978-1-59593-672-1. Available: <http://doi.acm.org/10.1145/1294325.1294327>. Citation on page 195.

____. Teaching Software Quality Assurance by Encouraging Student Contributions to an Open Source Web-based System for the Assessment of Programming Assignments. **SIGCSE Bull.**, v. 40, n. 3, p. 214–218, Jun. 2008. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1597849.1384329>. Citation on page 192.

GROSS, T. R. Breadth in depth: A 1st year introduction to parallel programming. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)**. New York, NY, USA: ACM, 2011. p. 435–440. ISBN 978-1-4503-0500-6. Citation on page 30.

GROVER, S.; PEA, R. Computational thinking in k-12: A review of the state of the field. **Educational Researcher**, v. 42, n. 1, p. 38–43, 2013. Available: <https://doi.org/10.3102/0013189X12463051>. Citation on page 19.

GUSTAFSON, D.; DWYER, M. Work in progress : the adversarial testing system. In: **30th Annual Frontiers in Education Conference. Building on A Century of Progress in Engineering Education. Conference Proceedings (IEEE Cat. No.00CH37135)**. [S.l.: s.n.], 2000. v. 1, p. T1C/19–T1C/20 vol.1. Citation on page 197.

GUZDIAL, M. A media computation course for non-majors. In: **Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03)**. New York, NY, USA: ACM, 2003. p. 104–108. ISBN 1-58113-672-2. Citation on page 37.

____. Education: Paving the way for computational thinking. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 8, p. 25–27, Aug. 2008. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/1378704.1378713>. Citation on page 19.

____. From science to engineering. **Commun. ACM**, ACM, New York, NY, USA, v. 54, n. 2, p. 37–39, Feb. 2011. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/1897816.1897831>. Citation on page 30.

____. Exploring hypotheses about media computation. In: **Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)**. New York, NY, USA: ACM, 2013. p. 19–26. ISBN 978-1-4503-2243-0. Citations on pages 20 and 107.

____. **What do I mean by Computing Education Research? The Social Science Perspective**. 2018. Available at <https://computinged.wordpress.com/2018/11/05/what-do-i-mean-by-computing-education-research-the-social-science-perspective/>. Citation on page 19.

HANKS, B.; FITZGERALD, S.; MCCAULEY, R.; MURPHY, L.; ZANDER, C. Pair programming in education: a literature review. **Computer Science Education**, v. 21, n. 2, p. 135–173, 2011. Citation on page 36.

HARRIS, J. A.; ADAMS, E. S.; HARRIS, N. L. Making program grading easier: But not totally automatic. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 20, n. 1, p. 248–261, Oct. 2004. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1040231.1040264>. Citation on page 198.

HELIOTIS, J.; ZANIBBI, R. Moving Away from Programming and Towards Computer Science in the CS First Year. **J. Comput. Sci. Coll.**, v. 26, n. 3, p. 115–125, Jan. 2011. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1859159.1859183>. Citation on page 191.

HELMICK, M. T. Interface-based Programming Assignments and Automatic Grading of Java Programs. In: **Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2007. (ITiCSE '07), p. 63–67. ISBN 978-1-59593-610-3. Available: <http://doi.acm.org/10.1145/1268784.1268805>. Citation on page 197.

HERNAN-LOSADA, I.; PAREJA-FLORES, C.; VELAZQUEZ-ITURBIDE, J. A. Testing-Based Automatic Grading: A Proposal from Bloom's Taxonomy. In: **2008 Eighth IEEE International Conference on Advanced Learning Technologies**. [S.l.: s.n.], 2008. p. 847–849. Citation on page 193.

HEROUT, P.; BRADA, P. Uml-test application for automated validation of students' uml class diagram. In: **2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)**. [S.l.: s.n.], 2016. p. 222–226. Citation on page 196.

HERTZ, M. 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). New York, NY, USA: ACM, 2010. p. 199–203. ISBN 978-1-4503-0006-3. Citation on page 28.

HERTZ, M.; FORD, S. M. Investigating factors of student learning in introductory courses. In: **Proceeding of the 44th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2013. (SIGCSE '13), p. 195–200. ISBN 978-1-4503-1868-6. Available: <http://doi.acm.org/10.1145/2445196.2445254>. Citations on pages 30 and 35.

HIGGINS, C.; SYMEONIDIS, P.; TSINTSIFAS, A. The marking system for coursemaster. In: **Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2002. (ITiCSE '02), p. 46–50. ISBN 1-58113-499-1. Available: <http://doi.acm.org/10.1145/544414.544431>. Citation on page 197.

HIGGINS, C. A.; GRAY, G.; SYMEONIDIS, P.; TSINTSIFAS, A. Automated assessment and experiences of teaching programming. **J. Educ. Resour. Comput.**, ACM, New York, NY, USA, v. 5, n. 3, Sep. 2005. ISSN 1531-4278. Available: <http://doi.acm.org/10.1145/1163405.1163410>. Citation on page 195.

HILTON, M.; JANZEN, D. S. 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**. New York, NY, USA: ACM, 2012. (ITiCSE '12), p. 93–98. ISBN 978-1-4503-1246-2. Available: <http://doi.acm.org/10.1145/2325296.2325322>. Citations on pages 101 and 192.

HORTON, D.; CRAIG, M.; CAMPBELL, J.; GRIES, P.; ZINGARO, D. Comparing outcomes in inverted and traditional cs1. In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)**. New York, NY, USA: ACM, 2014. p. 261–266. ISBN 978-1-4503-2833-3. Citation on page 35.

HORVATH, R. Software testing in introductory programming courses. In: **2012 IEEE 10th International Conference on Emerging eLearning Technologies and Applications (ICETA)**. [S.l.: s.n.], 2012. p. 133–134. Citation on page 193.

HU, H. H.; SHEPHERD, T. D. Using pogil to help students learn to program. **Trans. Comput. Educ.**, ACM, New York, NY, USA, v. 13, n. 3, p. 13:1–13:23, Aug. 2013. ISSN 1946-6226. Available: <http://doi.acm.org/10.1145/2499947.2499950>. Citation on page 35.

HULL, M. J.; POWELL, D.; KLEIN, E. Infandango: Automated Grading for Student Programming. In: **Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2011. (ITiCSE '11), p. 330–330. ISBN 978-1-4503-0697-3. Available: <http://doi.acm.org/10.1145/1999747.1999841>. Citation on page 198.

HUNDLEY, J. Imprinting Community College Computer Science Education with Software Engineering Principles: Work in Progress. In: **Proceedings of the 48th Annual Southeast Regional Conference**. New York, NY, USA: ACM, 2010. (ACM SE '10), p. 54:1–54:4. ISBN 978-1-4503-0064-3. Available: <http://doi.acm.org/10.1145/1900008.1900082>. Citation on page 195.

IHANTOLA, P. Creating and visualizing test data from programming exercises. **Informatics in Education**, 2007. Citation on page 197.

IHANTOLA, P.; AHONIEMI, T.; KARAVIRTA, V.; SEPPäLä, O. Review of recent systems for automatic assessment of programming assignments. In: **Proceedings of the 10th Koli Calling International Conference on Computing Education Research**. New York, NY, USA: ACM, 2010. (Koli Calling '10), p. 86–93. ISBN 978-1-4503-0520-4. Available: <http://doi.acm.org/10.1145/1930464.1930480>. Citation on page 197.

ISHIHARA, N.; FUNABIKI, N. A Proposal of Statement Fill-in-Blank Problem in Java Programming Learning Assistant System. In: **2015 IIAI 4th International Congress on Advanced Applied Informatics**. [S.l.: s.n.], 2015. p. 247–252. Citation on page 196.

ISOMOTTONEN, V.; LAPPALAINEN, V. Csi with games and an emphasis on tdd and unit testing: Piling a trend upon a trend. **ACM Inroads**, ACM, New York, NY, USA, v. 3, n. 3, p. 62–68, Sep. 2012. ISSN 2153-2184. Available: <http://doi.acm.org/10.1145/2339055.2339073>. Citations on pages 94, 98, 99, 103, 104, and 192.

ISONG, J. Developing an automated program checkers. In: **Proceedings of the Twelfth Annual CCSC South Central Conference on The Journal of Computing in Small Colleges**. USA: Consortium for Computing Sciences in Colleges, 2001. p. 218–224. Available: <http://dl.acm.org/citation.cfm?id=374678.374787>. Citation on page 194.

IZE, C.; POPE, C.; WEERASINGHE, A. On the ability to reason about program behaviour: a think-aloud study. In: **Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2017. (ITiCSE '17), p. 305–310. Citation on page 200.

JACKSON, D. A semi-automated approach to online assessment. In: **Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSEconference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2000. (ITiCSE '00), p. 164–167. ISBN 1-58113-207-7. Available: <http://doi.acm.org/10.1145/343048.343160>. Citation on page 197.

JANZEN, D.; SAIEDIAN, H. Test-driven Learning in Early Programming Courses. In: **Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2008. (SIGCSE '08), p. 532–536. ISBN 978-1-59593-799-5. Available: <http://doi.acm.org/10.1145/1352135.1352315>. Citations on pages 20, 36, 75, 86, 91, 94, 95, 96, 97, 98, 125, 140, and 192.

JANZEN, D. S.; CLEMENTS, J.; HILTON, M. An Evaluation of Interactive Test-driven Labs with WebIDE in CS0. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 1090–1098. ISBN 978-1-4673-3076-3. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486938>. Citations on pages 87, 130, and 196.

JANZEN, D. S.; SAIEDIAN, H. On the Influence of Test-Driven Development on Software Design. In: **19th Conference on Software Engineering Education Training (CSEET'06)**. [S.l.: s.n.], 2006. p. 141–148. Citations on pages 87, 88, 91, 92, 95, 125, 129, 131, and 194.

____. Test-driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In: **Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2006. (SIGCSE '06), p. 254–258. ISBN 1-59593-259-3. Available: <http://doi.acm.org/10.1145/1121341.1121419>. Citations on pages 86, 94, and 192.

____. A Leveled Examination of Test-Driven Development Acceptance. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 719–722. ISBN 0-7695-2828-7. Available: <http://dx.doi.org/10.1109/ICSE.2007.8>. Citations on pages 87, 95, 101, 102, 129, 133, 134, and 200.

JEDLITSCHKA, A.; CIOLKOWSKI, M.; PFAHL, D. Guide to advanced empirical software engineering. In: ____. London: Springer-Verlag, 2008. chap. Reporting Experiments in Software Engineering, p. 201–228. Citation on page 55.

JEZEK, P.; MALOHLAVA, M.; POP, T. Automated evaluation of regular lab assignments: A bittersweet experience? In: **2013 26th International Conference on Software Engineering Education and Training (CSEE&T)**. [S.l.: s.n.], 2013. p. 249–258. ISSN 1093-0175. Citations on pages 87, 90, 94, 98, 130, and 196.

JOHNSON, C. SpecCheck: Automated Generation of Tests for Interface Conformance. In: **Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2012. (ITiCSE '12), p. 186–191. ISBN 978-1-4503-1246-2. Available: <http://doi.acm.org/10.1145/2325296.2325343>. Citation on page 198.

JOHNSON, D. E. Itch: Individual testing of computer homework for scratch assignments. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 223–227. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844600>. Citation on page 198.

JONES, C. G. Test-driven Development Goes to School. **J. Comput. Sci. Coll.**, v. 20, n. 1, p. 220–231, Oct. 2004. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1040231.1040261>. Citation on page 194.

JONES, E. An experiential approach to incorporating software testing into the computer science curriculum. In: **31st Annual Frontiers in Education Conference**. [S.l.: s.n.], 2001. v. 2, p. F3D–7–F3D–11 vol.2. ISSN 0190-5848. Citation on page 191.

JONES, E. L. Grading Student Programs - a Software Testing Approach. In: **Proceedings of the Fourteenth Annual Consortium on Small Colleges Southeastern Conference**. USA: Consortium for Computing Sciences in Colleges, 2000. (CCSC '00), p. 185–192. Available: <http://dl.acm.org/citation.cfm?id=369340.369354>. Citation on page 194.

____. Software testing in the computer science curriculum – a holistic approach. In: **Proceedings of the Australasian Conference on Computing Education**. New York, NY, USA: ACM, 2000. (ACSE '00), p. 153–157. ISBN 1-58113-271-9. Available: <http://doi.acm.org/10.1145/359369.359392>. Citation on page 191.

____. The sprae framework for teaching software testing in the undergraduate curriculum. In: **Proceedings ADMI 2000**. [s.n.], 2000. Available: <https://pdfs.semanticscholar.org/f68a/ff6f5ef73f08e533b50da917751f340fef78.pdf>. Citation on page 192.

____. Integrating Testing into the Curriculum – Arsenic in Small Doses. In: **Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2001. (SIGCSE '01), p. 337–341. ISBN 1-58113-329-4. Available: <http://doi.acm.org/10.1145/364447.364617>. Citations on pages 20, 75, 133, 140, and 191.

JONES, E. L.; ALLEN, C. S. Repositories for CS Courses: An Evolutionary Tale. In: **Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2003. (ITiCSE '03), p. 119–123. ISBN 1-58113-672-2. Available: <http://doi.acm.org/10.1145/961511.961546>. Citation on page 197.

JOSHI, G.; DESAI, P. Building Software Testing Skills in Undergraduate Students Using Spiral Model Approach. In: **2016 IEEE Eighth International Conference on Technology for Education (T4E)**. [S.l.: s.n.], 2016. p. 244–245. Citation on page 192.

JOY, M.; GRIFFITHS, N.; BOYATT, R. The boss online submission and assessment system. **J. Educ. Resour. Comput.**, ACM, New York, NY, USA, v. 5, n. 3, Sep. 2005. ISSN 1531-4278. Available: <http://doi.acm.org/10.1145/1163405.1163407>. Citation on page 198.

JUEDES, D. W. Web-based grading: further experiences and student attitudes. In: **Proceedings Frontiers in Education 35th Annual Conference**. [S.l.: s.n.], 2005. p. F4E–18. Citation on page 198.

JURISTO, N.; MORENO, A. M. **Basics of Software Engineering Experimentation**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2001. ISBN 1441950117, 9781441950116. Citations on pages 11, 13, 23, 39, 41, 42, 43, 47, 48, 49, 54, 60, 87, and 110.

JURISTO, N.; MORENO, A. M.; VEGAS, S. Reviewing 25 years of testing technique experiments. **Empirical Software Engineering**, Kluwer Academic Publishers, Hingham, MA, USA, v. 9, n. 1-2, p. 7–44, Mar. 2004. ISSN 1382-3256. Citation on page 40.

KARAVIRTA, V.; IHANTOLA, P. Serverless Automatic Assessment of Javascript Exercises. In: **Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2010. (ITiCSE '10), p. 303–303. ISBN 978-1-60558-820-9. Available: <http://doi.acm.org/10.1145/1822090.1822179>. Citation on page 198.

KART, M. Test First Programming, Design by Contract, and Intriguing Coursework: Ingredients for Increasing Student Engagement. **J. Comput. Sci. Coll.**, v. 28, n. 4, p. 35–41, Apr. 2013. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2458539.2458545>. Citation on page 192.

KAUSHAL, R.; SINGH, A. Automated evaluation of programming assignments. In: **2012 IEEE International Conference on Engineering Education: Innovative Practices and Future Trends (AICERA)**. [S.l.: s.n.], 2012. p. 1–5. Citation on page 196.

KEEFE, K.; SHEARD, J.; DICK, M. Adopting xp practices for teaching object oriented programming. In: **Proceedings of the 8th Australasian Conference on Computing Education - Volume 52**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006. (ACE '06), p. 91–100. ISBN 1-920682-34-1. Available: <http://dl.acm.org/citation.cfm?id=1151869.1151882>. Citation on page 195.

KINNUNEN, P.; MALMI, L. Why students drop out cs1 course? In: **Proceedings of the Second International Workshop on Computing Education Research (ICER '06)**. New York, NY, USA: ACM, 2006. p. 97–108. ISBN 1-59593-494-4. Citation on page 31.

KITCHENHAM, B.; BUDGEN, D.; BRERETON, P.; WOODALL, P. An investigation of software engineering curricula. **Journal of Systems and Software**, v. 74, n. 3, p. 325 – 335, 2005. Citation on page 64.

KITCHENHAM, B.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. [S.l.], 2007. Citation on page 23.

KITCHENHAM, B. A.; PFLEEGER, S. L. Personal opinion surveys. In: ____. **Guide to Advanced Empirical Software Engineering**. London: Springer London, 2008. p. 63–92. Citation on page 64.

KITCHENHAM, B. A.; PFLEEGER, S. L.; PICKARD, L. M.; JONES, P. W.; HOAGLIN, D. C.; EMAM, K. E.; ROSENBERG, J. Preliminary guidelines for empirical research in software engineering. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 8, p. 721–734, 2002. Citation on page 40.

KITCHENHAM, B. A.; TRAVASSOS, G. H.; MAYRHAUSER, A. von; NIESSINK, F.; SCHNEIDEWIND, N. F.; SINGER, J.; TAKADA, S.; VEHVILAINEN, R.; YANG, H. Towards an ontology of software maintenance. **Journal of Software Maintenance**, John Wiley & Sons, Inc., New York, NY, USA, v. 11, n. 6, p. 365–389, Nov. 1999. ISSN 1040-550X. Available: <http://dx.doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W>. Citation on page 55.

KOLIKANT, Y. B.-D. Students' Alternative Standards for Correctness. In: **Proceedings of the First International Workshop on Computing Education Research**. New York, NY, USA: ACM, 2005. (ICER '05), p. 37–43. ISBN 1-59593-043-4. Available: <http://doi.acm.org/10.1145/1089786.1089790>. Citation on page 200.

KOLIKANT, Y. B.-D.; MUSSAI, M. ?so my program doesn?t run!? definition, origins, and practical expressions of students? (mis)conceptions of correctness. **Computer Science Education**, Routledge, v. 18, n. 2, p. 135–151, 2008. Citation on page 200.

KOLLING, M.; QUIG, B.; PATTERSON, A.; ROSENBERG, J. The bluej system and its pedagogy. **Computer Science Education**, Routledge, v. 13, n. 4, p. 249–268, 2003. Available: <http://www.tandfonline.com/doi/abs/10.1076/csed.13.4.249.17496>. Citation on page 192.

KOULOURI, T.; LAURIA, S.; MACREDIE, R. D. Teaching introductory programming: A quantitative evaluation of different approaches. **ACM Transactions on Computing Education (TOCE)**, ACM, New York, NY, USA, v. 14, n. 4, p. 26:1–26:28, Dec. 2014. ISSN 1946-6226. Citations on pages 21, 30, 35, and 139.

KRUSCHE, S.; SEITZ, A. Artemis: An automatic assessment management system for interactive learning. In: **Proceedings of the 49th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2018. (SIGCSE '18), p. 284–289. ISBN 978-1-4503-5103-4. Available: <http://doi.acm.org/10.1145/3159450.3159602>. Citations on pages 91, 98, 100, 103, 135, and 197.

KUSSMAUL, C. L. Scaffolding for Multiple Assignment Projects in CS1 and CS2. In: **Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications**. New York, NY, USA: ACM, 2008. (OOPSLA Companion '08), p. 873–876. ISBN 978-1-60558-220-7. Available: <http://doi.acm.org/10.1145/1449814.1449890>. Citation on page 194.

KYRILOV, A.; NOELLE, D. C. Do Students Need Detailed Feedback on Programming Exercises and Can Automated Assessment Systems Provide It? **J. Comput. Sci. Coll.**, v. 31, n. 4, p. 115–121, Apr. 2016. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2904127.2904147>. Citation on page 196.

LAHTINEN, E.; ALA-MUTKA, K.; JARVINEN, H.-M. A study of the difficulties of novice programmers. In: **Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '05)**. New York, NY, USA: ACM, 2005. p. 14–18. ISBN 1-59593-024-8. Citations on pages 13, 31, and 33.

LAKANEN, A.-J.; LAPPALAINEN, V.; ISOMöTTöNEN, V. Revisiting rainfall to explore exam questions and performance on cs1. In: **Proceedings of the 15th Koli Calling Conference on Computing Education Research**. New York, NY, USA: ACM, 2015. (Koli Calling '15), p. 40–49. ISBN 978-1-4503-4020-5. Available: <http://doi.acm.org/10.1145/2828959.2828970>. Citation on page 194.

LAPPALAINEN, V.; ITKONEN, J.; ISOMOTTONEN, V.; KOLLANUS, S. ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming. In: **Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2010. (ITiCSE '10), p. 63–67. ISBN 978-1-60558-820-9. Available: <http://doi.acm.org/10.1145/1822090.1822110>. Citation on page 195.

LEAL, J. P.; SILVA, F. Mooshak: a web-based multi-site programming contest system. **Software: Practice and Experience**, Wiley Online Library, v. 33, n. 6, p. 567–581, 5 2003. ISSN 1097-024X. Available: <http:https://doi.org/10.1002/spe.522>. Citation on page 195.

____. A new learning paradigm: Competition supported by technology. In: ____. [S.l.]: Sello Editorial, 2010. chap. Using Mooshak as a competitive learning tool, p. 91–106. Citation on page 198.

LEE, Y.; MAREPALLI, D. B.; YANG, J. Teaching test-drive development using dojo. **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 32, n. 4, p. 106–112, Apr. 2017. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=3055338.3079049>. Citations on pages 92, 103, and 192.

LEMOS, O. A. L.; FERRARI, F. C.; SILVEIRA, F. F.; GARCIA, A. Experience report: Can software testing education lead to more reliable code? In: **2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2015. p. 359–369. Citations on pages 86, 88, 89, 90, 96, 128, and 199.

LEMOS, O. A. L.; SILVEIRA, F. F.; FERRARI, F. C.; GARCIA, A. The impact of Software Testing education on code reliability: An empirical assessment. **Journal of Systems and Software**, p. –, 2017. ISSN 0164-1212. Available: <http://www.sciencedirect.com/science/article/pii/S0164121217300419>. Citations on pages 86, 88, 89, 90, 96, 128, and 199.

LESKA, C. Testing across the curriculum: Square one! **J. Comput. Sci. Coll.**, Consortium for Computing Sciences in Colleges, USA, v. 19, n. 5, p. 163–169, May 2004. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1060081.1060103>. Citation on page 191.

LESKA, C.; RABUNG, J. Refactoring the CS1 Course. **J. Comput. Sci. Coll.**, v. 20, n. 3, p. 6–18, Feb. 2005. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1040196.1040199>. Citation on page 192.

LETHBRIDGE, T. C. What knowledge is important to a software professional? **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 33, n. 5, p. 44–50, 2000. ISSN 0018-9162. Citations on pages 19, 64, and 66.

LETHBRIDGE, T. C.; DIAZ-HERRERA, J.; LEBLANC, R. J. J.; THOMPSON, J. B. Improving software practice through education: Challenges and future trends. In: **Future of Software Engineering (FOSE '07)**. [S.l.: s.n.], 2007. p. 12–28. Citations on pages 19, 73, and 74.

LI, J. J.; MORREALE, P. Enhancing CS1 Curriculum with Testing Concepts: A Case Study. **J. Comput. Sci. Coll.**, v. 31, n. 3, p. 36–43, Jan. 2016. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2835377.2835384>. Citations on pages 86, 96, and 192.

LIPPE, T. v. d.; SMITH, T.; PELSMAEKER, D.; VISSER, E. A scalable infrastructure for teaching concepts of programming languages in scala with weblab: An experience report. In: **Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala**. New York, NY, USA: ACM, 2016. (SCALA 2016), p. 65–74. ISBN 978-1-4503-4648-1. Available: <http://doi.acm.org/10.1145/2998392.2998402>. Citation on page 198.

LISHINSKI, A.; GOOD, J.; SANDS, P.; YADAV, A. Methodological rigor and theoretical foundations of cs education research. In: **Proceedings of the 2016 ACM Conference on International Computing Education Research**. New York, NY, USA: ACM, 2016. (ICER '16), p. 161–169. ISBN 978-1-4503-4449-4. Available: <http://doi.acm.org/10.1145/2960310.2960328>. Citations on pages 21, 22, 23, 123, and 139.

LISTER, R. Ten years after the mccracken working group. **ACM Inroads**, ACM, New York, NY, USA, v. 2, n. 4, p. 18–19, Dec. 2011. ISSN 2153-2184. Available: <http://doi.acm.org/10.1145/2038876.2038882>. Citations on pages 25 and 31.

LISTER, R.; ADAMS, E. S.; FITZGERALD, S.; FONE, W.; HAMER, J.; LINDHOLM, M.; MCCARTNEY, R.; MOSTROM, J. E.; SANDERS, K.; SEPPALA, O.; SIMON, B.; THOMAS, L. A multi-national study of reading and tracing skills in novice programmers. In: **Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2004. (ITiCSE-WGR '04), p. 119–150. Available: <http://doi.acm.org/10.1145/1044550.1041673>. Citations on pages 20 and 25.

LLANA, L.; MARTIN-MARTIN, E.; PAREJA-FLORES, C. FLOP, a Free Laboratory of Programming. In: **Proceedings of the 12th Koli Calling International Conference on Computing Education Research**. New York, NY, USA: ACM, 2012. (Koli Calling '12), p. 93–99. ISBN 978-1-4503-1795-5. Available: <http://doi.acm.org/10.1145/2401796.2401807>. Citation on page 198.

LUXTON-REILLY, A. Learning to program is easy. In: **Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2016. (ITiCSE '16), p. 284–289. ISBN 978-1-4503-4231-5. Available: <http://doi.acm.org/10.1145/2899415.2899432>. Citations on pages 25 and 35.

LUXTON-REILLY, A.; BECKER, B. A.; CAO, Y.; MCDERMOTT, R.; MIROLO, C.; MüHLING, A.; PETERSEN, A.; SANDERS, K.; SIMON; WHALLEY, J. Developing assessments to determine mastery of programming fundamentals. In: **Proceedings of the 2017 ITiCSE Conference on Working Group Reports**. New York, NY, USA: ACM, 2017. (ITiCSE-WGR '17), p. 47–69. ISBN 978-1-4503-5627-5. Available: <http://doi.acm.org/10.1145/3174781.3174784>. Citation on page 199.

LUXTON-REILLY, A.; DENNY, P.; KIRK, D.; TEMPERO, E.; YU, S.-Y. On the differences between correct student solutions. In: **Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2013. (ITiCSE

'13), p. 177–182. ISBN 978-1-4503-2078-8. Available: <http://doi.acm.org/10.1145/2462476.2462505>. Citation on page 199.

LUXTON-REILLY, A.; SIMON; ALBLUWI, I.; BECKER, B. A.; GIANNAKOS, M.; KUMAR, A. N.; OTT, L.; PATERSON, J.; SCOTT, M. J.; SHEARD, J.; SZABO, C. A review of introductory programming research 2003–2017. In: **Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2018. (ITiCSE 2018), p. 342–343. ISBN 978-1-4503-5707-4. Available: <http://doi.acm.org/10.1145/3197091.3205841>. Citations on pages 19 and 25.

LYE, S. Y.; KOH, J. H. L. Review on teaching and learning of computational thinking through programming: What is next for k-12? **Computers in Human Behavior**, v. 41, p. 51 – 61, 2014. ISSN 0747-5632. Available: <http://www.sciencedirect.com/science/article/pii/S0747563214004634>. Citation on page 19.

MADEJA, M.; PORUBAN, J. Automatic assessment of assignments for android application programming courses. In: **2017 IEEE 14th International Scientific Conference on Informatics**. [S.l.: s.n.], 2017. p. 232–237. Citations on pages 100 and 197.

MAJOR, L.; KYRIACOU, T.; BRERETON, O. Systematic literature review: Teaching novices programming using robots. In: **15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011)**. [S.l.: s.n.], 2011. p. 21–30. Citation on page 38.

MALMI, L.; SHEARD, J.; SIMON; BEDNARIK, R.; HELMINEN, J.; KORHONEN, A.; MYLLER, N.; SORVA, J.; TAHERKHANI, A. Characterizing research in computing education: A preliminary analysis of the literature. In: **Proceedings of the Sixth International Workshop on Computing Education Research**. New York, NY, USA: ACM, 2010. (ICER '10), p. 3–12. ISBN 978-1-4503-0257-9. Available: <http://doi.acm.org/10.1145/1839594.1839597>. Citations on pages 21, 23, 123, and 139.

MALMI, L.; SHEARD, J.; SIMON; BEDNARIK, R.; HELMINEN, J.; KINNUNEN, P.; KORHONEN, A.; MYLLER, N.; SORVA, J.; TAHERKHANI, A. Theoretical underpinnings of computing education research: What is the evidence? In: **Proceedings of the Tenth Annual Conference on International Computing Education Research**. New York, NY, USA: ACM, 2014. (ICER '14), p. 27–34. ISBN 978-1-4503-2755-8. Available: <http://doi.acm.org/10.1145/2632320.2632358>. Citations on pages 21, 22, 123, and 139.

MARCOS-ABED, J. Learning Computer Programming: A Study of the Effectiveness of a COAC#. In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education**. New York, NY, USA: ACM, 2014. (ITiCSE '14), p. 333–333. ISBN 978-1-4503-2833-3. Available: <http://doi.acm.org/10.1145/2591708.2602652>. Citation on page 198.

____. Using a COAC# for CS1. In: **Proceedings of the Western Canadian Conference on Computing Education**. New York, NY, USA: ACM, 2014. (WCCCE '14), p. 10:1–10:3. ISBN 978-1-4503-2899-9. Available: <http://doi.acm.org/10.1145/2597959.2597971>. Citation on page 196.

MARKHAM, S. A.; KING, K. N. 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)**. New York, NY, USA: ACM, 2010. p. 204–208. ISBN 978-1-60558-820-9. Citations on pages 30 and 135.

MARRERO, W.; SETTLE, A. Testing First: Emphasizing Testing in Early Programming Courses. In: **Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2005. (ITiCSE '05), p. 4–8. ISBN 1-59593-024-8. Available: <http://doi.acm.org/10.1145/1067445.1067451>. Citation on page 193.

MATTHIES, C.; TREFFER, A.; UFLACKER, M. Prof. ci: Employing continuous integration services and github workflows to teach test-driven development. In: **2017 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2017. p. 1–8. Citations on pages 96, 98, 101, and 192.

MCCARTNEY, R.; BOUSTEDT, J.; ECKERDAL, A.; SANDERS, K.; ZANDER, C. Can first-year students program yet?: A study revisited. In: **Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research**. New York, NY, USA: ACM, 2013. (ICER '13), p. 91–98. ISBN 978-1-4503-2243-0. Available: <http://doi.acm.org/10.1145/2493394.2493412>. Citations on pages 20, 22, 32, 34, and 38.

MCCRACKEN, M.; ALMSTRUM, V.; DIAZ, D.; GUZDIAL, M.; HAGAN, D.; KOLIKANT, Y. B.-D.; LAXER, C.; THOMAS, L.; UTTING, I.; WILUSZ, T. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 33, n. 4, p. 125–180, Dec. 2001. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/572139.572181>. Citations on pages 11, 13, 20, 22, 25, 31, 32, 34, and 38.

MCDOWELL, C.; WERNER, L.; BULLOCK, H.; FERNALD, J. The effects of pair-programming on performance in an introductory programming course. In: **33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)**. New York, NY, USA: ACM, 2002. p. 38–42. ISBN 1-58113-473-8. Citations on pages 36 and 59.

MCDOWELL, C.; WERNER, L.; BULLOCK, H. E.; FERNALD, J. The impact of pair programming on student performance, perception and persistence. In: **25th International Conference on Software Engineering (ICSE '03)**. Washington, DC, USA: IEEE Computer Society, 2003. p. 602–607. ISBN 0-7695-1877-X. Citation on page 36.

MCGILL, M. M. Learning to program with personal robots: Influences on student motivation. **ACM Transactions on Computing Education**, ACM, New York, NY, USA, v. 12, n. 1, p. 4:1–4:32, Mar. 2012. ISSN 1946-6226. Citation on page 38.

MENDONCA, A.; GUERRERO, D.; COSTA, E. An approach for problem specification and its application in an Introductory Programming Course. In: **2009 39th IEEE Frontiers in Education Conference**. [S.l.: s.n.], 2009. p. 1–6. Citation on page 194.

MENDONCA, A.; OLIVEIRA, C. d.; GUERRERO, D.; COSTA, E. Difficulties in solving ill-defined problems: A case study with introductory computer programming students. In: **2009 39th IEEE Frontiers in Education Conference**. [S.l.: s.n.], 2009. p. 1–6. Citation on page 200.

MENDONCA, M.; CRUZES, D.; DIAS, J.; OLIVEIRA, M. C. F. de. Using observational pilot studies to test and improve lab packages. In: **Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)**. New York, NY, USA: ACM, 2006. p. 48–57. ISBN 1-59593-218-6. Citation on page 51.

MIDDLETON, D. Developing Students' Testing Skills: Distinguishing Functions. **J. Comput. Sci. Coll.**, v. 28, n. 5, p. 73–74, May 2013. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2458569.2458583>. Citation on page 194.

____. Developing Students' Testing Skills: Covering Space: Nifty Assignment. **J. Comput. Sci. Coll.**, v. 30, n. 5, p. 29–31, May 2015. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2752981.2752988>. Citation on page 194.

MILLER, K. W. Test Driven Development on the Cheap: Text Files and Explicit Scaffolding. **J. Comput. Sci. Coll.**, v. 20, n. 2, p. 181–189, Dec. 2004. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1040151.1040172>. Citation on page 192.

MISSIROLI, M.; RUSSO, D.; CIANCARINI, P. Teaching test-first programming: Assessment and solutions. In: **2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)**. [S.l.: s.n.], 2017. v. 1, p. 420–425. ISSN 0730-3157. Citations on pages 102, 104, and 194.

MORENO, A. M.; SANCHEZ-SEGURA, M.-I.; MEDINA-DOMINGUEZ, F.; CARVAJAL, L. Balancing software engineering education and industrial needs. **Journal of Systems and Software**, v. 85, n. 7, p. 1607 – 1620, 2012. Software Ecosystems. Citation on page 63.

MORISIO, M.; TORCHIANO, M.; ARGENTIERI, G. Assessing quantitatively a programming course. In: **10th International Symposium on Software Metrics, 2004. Proceedings.** [S.l.: s.n.], 2004. p. 326–336. Citations on pages 88, 90, 91, 95, 96, 104, and 199.

MORRIS, D. S. Automatic grading of student's programming assignments: an interactive process and suite of programs. In: **33rd Annual Frontiers in Education, 2003. FIE 2003.** [S.l.: s.n.], 2003. v. 3, p. S3F–1–6 vol.3. ISSN 0190-5848. Citation on page 197.

MORRISON, P. A security practices evaluation framework. In: **Proceedings of the 37th International Conference on Software Engineering - Volume 2**. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 935–938. Available: <http://dl.acm.org/citation.cfm?id=2819009.2819218>. Citations on pages 23, 55, and 60.

MOURA, I. C.; HATTUM-JANSSEN, N. van. Teaching a cs introductory course: An active approach. **Computers & Education**, Elsevier Science Ltd., Oxford, UK, UK, v. 56, n. 2, p. 475–483, Feb. 2011. ISSN 0360-1315. Citation on page 35.

MULLER, M. M.; TICHY, W. F. Case study: extreme programming in a university environment. In: **Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)**. [S.l.: s.n.], 2001. p. 537–544. Citation on page 59.

MUNSON, J. C. **Software Engineering Measurement**. Boca Raton, FL, USA: CRC Press, Inc., 2002. ISBN 0849315034. Citation on page 87.

MURPHY, L.; LEWANDOWSKI, G.; MCCAULEY, R.; SIMON, B.; THOMAS, L.; ZANDER, C. Debugging: The good, the bad, and the quirky – a qualitative analysis of novices' strategies. In: **Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2008. (SIGCSE '08), p. 163–167. ISBN 978-1-59593-799-5. Available: <http://doi.acm.org/10.1145/1352135.1352191>. Citation on page 195.

MURPHY, M. C.; YILDIRIM, B. Work in progress - testing right from the start. In: **2007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports**. [S.l.: s.n.], 2007. p. F1H–25–F1H–26. Citation on page 197.

NAGAPPAN, N.; WILLIAMS, L.; FERZLI, M.; WIEBE, E.; YANG, K.; MILLER, C.; BALIK, S. Improving the cs1 experience with pair programming. **SIGCSE Bulletin**, ACM, New York, NY, USA, v. 35, n. 1, p. 359–362, Jan. 2003. ISSN 0097-8418. Citation on page 36.

NAPS, T. L.; ROSSLING, G.; ALMSTRUM, V.; DANN, W.; FLEISCHER, R.; HUNDHAUSEN, C.; KORHONEN, A.; MALMI, L.; MCNALLY, M.; RODGER, S.; VELAZQUEZ-ITURBIDE, J. A. Exploring the role of visualization and engagement in computer science education. In: **Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '02)**. New York, NY, USA: ACM, 2002. p. 131–152. Citation on page 37.

NAWROCKI, J.; WOJCIECHOWSKI, A. Experimental evaluation of pair programming. In: **proc. European Software Control and Metrics (Escom)**. [S.l.: s.n.], 2001. Citation on page 59.

NELSON, G. L.; KO, A. J. On use of theory in computing education research. In: **Proceedings of the 14th International Workshop on Computing Education Research**. New York, NY, USA: ACM, 2018. (ICER '18), p. 1–10. ISBN 978-1-4503-0257-9. Available: <https://doi.org/10.1145/3230977.3230992>. Citation on page 22.

NETO, V. L.; COELHO, R.; LEITE, L.; GUERRERO, D. S.; MENDONCA, A. P. POPT: A Problem-oriented Programming and Testing Approach for Novice Students. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 1099–1108. ISBN 978-1-4673-3076-3. Available: <http://dl.acm.org/citation.cfm?id=2486788.2486939>. Citations on pages 87, 90, 96, 98, 101, 129, 134, and 194.

NG, S. P.; MURNANE, T.; REED, K.; GRANT, D.; CHEN, T. Y. A preliminary survey on software testing practices in Australia. In: **Proceedings of the 2004 Australian Software Engineering Conference**. Washington, DC, USA: IEEE Computer Society, 2004. (ASWEC '04), p. 116–. ISBN 0-7695-2089-8. Citation on page 64.

NINO, J. Introducing API Design Principles in CS2. **J. Comput. Sci. Coll.**, v. 24, n. 4, p. 109–116, Apr. 2009. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1516546.1516566>. Citation on page 194.

NISHIMURA, T.; KAWASAKI, S.; TOMINAGA, H. Monitoring system of student situation in introductory C programming exercise with a contest style. In: **2011 International Conference on Information Technology Based Higher Education and Training**. [S.l.: s.n.], 2011. p. 1–6. Citations on pages 98 and 196.

NORDQUIST, P. Providing Accurate and Timely Feedback by Automatically Grading Student Programming Labs. **J. Comput. Sci. Coll.**, v. 23, n. 2, p. 16–23, Dec. 2007. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1292428.1292432>. Citation on page 198.

NORTH, D. **Introducing BDD**. 2006. Available at https://dannorth.net/introducing-bdd. Citation on page 70.

NOSEK, J. T. The case for collaborative programming. **Communications of the ACM**, ACM, New York, NY, USA, v. 41, n. 3, p. 105–108, Mar. 1998. ISSN 0001-0782. Citation on page 59.

O'BRIEN, C.; GOLDMAN, M.; MILLER, R. C. Java Tutor: Bootstrapping with Python to Learn Java. In: **Proceedings of the First ACM Conference on Learning Scale Conference**. New York, NY, USA: ACM, 2014. (L@S '14), p. 185–186. ISBN 978-1-4503-2669-8. Available: <http://doi.acm.org/10.1145/2556325.2567873>. Citation on page 198.

ODEKIRK-HASH, E.; ZACHARY, J. L. Automated feedback on programs means students need less help from teachers. In: **Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2001. (SIGCSE '01), p. 55–59. ISBN 1-58113-329-4. Available: <http://doi.acm.org/10.1145/364447.364537>. Citations on pages 87, 94, 100, and 196.

OLAN, M. Unit testing: Test early, test often. **J. Comput. Sci. Coll.**, v. 19, n. 2, p. 319–328, Dec. 2003. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=948785.948830>. Citation on page 192.

OLIVEIRA, R. A. P.; OLIVEIRA, L. B. R.; CAFEO, B. B. P.; DURELLI, V. H. S. Evaluation and assessment of effects on exploring mutation testing in programming courses. In: **2015 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2015. p. 1–9. Citations on pages 86, 94, 96, and 192.

ORSO, A.; HARROLD, M. J.; ROSENBLUM, D.; ROTHERMEL, G.; SOFFA, M. L.; DO, H. Using component metacontent to support the regression testing of component-based software. In: **Proceedings IEEE International Conference on Software Maintenance**. [S.l.: s.n.], 2001. p. 716–725. ISSN 1063-6773. Citation on page 119.

PAPE, S.; FLAKE, J.; BECKMANN, A.; JURJENS, J. STAGE: A Software Tool for Automatic Grading of Testing Exercises: Case Study Paper. In: **Proceedings of the 38th International Conference on Software Engineering Companion**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 491–500. ISBN 978-1-4503-4205-6. Available: <http://doi.acm.org/10.1145/2889160.2889203>. Citation on page 199.

PARODI, E.; MATALONGA, S.; MACCHI, D.; SOLARI, M. Comparing technical debt in student exercises using test driven development, test last and ad hoc programming. In: **2016 XLII Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2016. p. 1–10. Citations on pages 87, 91, 101, 129, and 194.

PARRISH, A.; CORDES, D.; DIXON, B.; MCGREGOR, J. Class Development and Testing in the Small. In: **Proceedings of the 38th Annual on Southeast Regional Conference**. New York, NY, USA: ACM, 2000. (ACM-SE 38), p. 139–145. ISBN 1-58113-250-6. Available: <http://doi.acm.org/10.1145/1127716.1127750>. Citation on page 195.

PATTERSON, A.; KöLLING, M.; ROSENBERG, J. Introducing unit testing with bluej. In: **Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2003. (ITiCSE '03), p. 11–15. ISBN 1-58113-672-2. Available: <http://doi.acm.org/10.1145/961511.961518>. Citation on page 197.

PAUL, J. Test-driven Approach in Programming Pedagogy. **J. Comput. Sci. Coll.**, v. 32, n. 2, p. 53–60, Dec. 2016. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=3015063.3015072>. Citations on pages 104 and 194.

PEARCE, J. L.; NAKAZAWA, M.; HEGGEN, S. Improving Problem Decomposition Ability in CS1 Through Explicit Guided Inquiry-based Instruction. **J. Comput. Sci. Coll.**, v. 31, n. 2, p. 135–144, Dec. 2015. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2831432.2831453>. Citation on page 195.

PEARS, A.; MALMI, L. Values and objectives in computing education research. **Trans. Comput. Educ.**, ACM, New York, NY, USA, v. 9, n. 3, p. 15:1–15:6, Sep. 2009. ISSN 1946-6226. Available: <http://doi.acm.org/10.1145/1594399.1594400>. Citations on pages 21, 123, and 139.

PEARS, A.; SEIDMAN, S.; MALMI, L.; MANNILA, L.; ADAMS, E.; BENNEDSEN, J.; DEVLIN, M.; PATERSON, J. 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)**. New York, NY, USA: ACM, 2007. p. 204–223. Available: <http://doi.acm.org/10.1145/1345443.1345441>. Citations on pages 25, 35, and 37.

PETERSEN, A.; CRAIG, M.; CAMPBELL, J.; TAFLIOVICH, A. Revisiting why students drop cs1. In: **Proceedings of the 16th Koli Calling International Conference on Computing Education Research**. New York, NY, USA: ACM, 2016. (Koli Calling '16), p. 71–80. ISBN 978-1-4503-4770-9. Available: <http://doi.acm.org/10.1145/2999541.2999552>. Citations on pages 20 and 31.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: **Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08)**. Swinton, UK, UK: British Computer Society, 2008. p. 68–77. Citations on pages 23, 76, and 78.

PETIT, J.; GIMENEZ, O.; ROURA, S. Jutge.Org: An Educational Programming Judge. In: **Proceedings of the 43rd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2012. (SIGCSE '12), p. 445–450. ISBN 978-1-4503-1098-7. Available: <http://doi.acm.org/10.1145/2157136.2157267>. Citation on page 195.

PHAM, R.; KIESLING, S.; LISKIN, O.; SINGER, L.; SCHNEIDER, K. Enablers, inhibitors, and perceptions of testing in novice software teams. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. [S.l.]: Association for Computing Machinery, 2014. v. 16-21-November-2014, p. 30–40. ISBN 978-1-4503-3056-5. Citation on page 200.

PIETERSE, V. Automated assessment of programming assignments. In: **Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research**. Open Univ., Heerlen, The Netherlands, The Netherlands: Open Universiteit, Heerlen, 2013. (CSERC '13), p. 4:45–4:56. Available: <http://dl.acm.org/citation.cfm?id=2541917.2541921>. Citation on page 196.

PIETERSE, V.; LIEBENBERG, J. Automatic vs manual assessment of programming tasks. In: **Proceedings of the 17th Koli Calling International Conference on Computing Education Research**. New York, NY, USA: ACM, 2017. (Koli Calling '17), p. 193–194. ISBN 978-1-4503-5301-4. Available: <http://doi.acm.org/10.1145/3141880.3141912>. Citations on pages 87, 97, 99, 134, and 193.

PIETRIKOVA, E.; JUHAR, J.; STASTNA, J. Towards automated assessment in game-creative programming courses. In: **2015 13th International Conference on Emerging eLearning Technologies and Applications (ICETA)**. [S.l.: s.n.], 2015. p. 1–6. Citation on page 198.

POLITZ, J. G.; COLLARD, J. M.; GUHA, A.; FISLER, K.; KRISHNAMURTHI, S. The Sweep: Essential Examples for In-Flow Peer Review. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 243–248. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844626>. Citations on pages 93, 103, 135, and 192.

POLITZ, J. G.; KRISHNAMURTHI, S.; FISLER, K. In-flow peer-review of tests in test-first programming. In: **Proceedings of the Tenth Annual Conference on International Computing Education Research**. New York, NY, USA: ACM, 2014. (ICER '14), p. 11–18. ISBN 978-1-4503-2755-8. Citations on pages 93, 98, 103, 135, and 192.

POLITZ, J. G.; PATTERSON, D.; KRISHNAMURTHI, S.; FISLER, K. Captainteach: Multi-stage, in-flow peer review for programming assignments. In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education**. New York, NY, USA: ACM, 2014. (ITiCSE '14), p. 267–272. ISBN 978-1-4503-2833-3. Citations on pages 95 and 196.

POZENEL, M.; FURST, L.; MAHNIC, V. Introduction of the automated assessment of homework assignments in a university-level programming course. In: **2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**. [S.l.: s.n.], 2015. p. 761–766. Citation on page 198.

PRIBELA, I.; PRACNER, D.; BUDIMAC, Z.; GRBAC, T. Tool for testing bad student programs. In: Z., G. T. B. (Ed.). **CEUR Workshop Proceedings**. CEUR-WS, 2014. v. 1266, p. 67–74. ISBN 978-86-7031-374-3. Available: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84909952186&partnerID=40&md5=102efb7a2488a140f9900c76fb587c09>. Citation on page 198.

PROULX, V. Introductory Computing: The Design Discipline. In: KALA, I.; MITTERMEIR, R. T. (Ed.). **Informatics in Schools. Contributing to 21st Century Education**. Springer Berlin Heidelberg, 2011, (Lecture Notes in Computer Science, v. 7013). p. 177–188. ISBN 978-3-642-24721-7. Available: <http://dx.doi.org/10.1007/978-3-642-24722-4_16>. Citation on page 193.

PROULX, V. K. Test-driven Design for Introductory OO Programming. In: **Proceedings of the 40th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2009. (SIGCSE '09), p. 138–142. ISBN 978-1-60558-183-5. Available: <http://doi.acm.org/10.1145/1508865.1508919>. Citation on page 192.

PROULX, V. K.; JOSSEY, W. Unit Test Support for Java via Reflection and Annotations. In: **Proceedings of the 7th International Conference on Principles and Practice of Programming in Java**. New York, NY, USA: ACM, 2009. (PPPJ '09), p. 49–56. ISBN 978-1-60558-598-7. Available: <http://doi.acm.org/10.1145/1596655.1596663>. Citation on page 195.

____. Unit Testing in Java. In: **Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2009. (ITiCSE '09), p. 349–349. ISBN 978-1-60558-381-5. Available: <http://doi.acm.org/10.1145/1562877.1562990>. Citation on page 197.

PROULX, V. K.; RASALA, R. Java io and testing made simple. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 36, n. 1, p. 161–165, Mar. 2004. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1028174.971358>. Citation on page 197.

RADERMACHER, A.; WALIA, G. Gaps between industry expectations and the abilities of graduates. In: **Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)**. New York, NY, USA: ACM, 2013. p. 525–530. ISBN 978-1-4503-1868-6. Citations on pages 19, 63, 64, 67, 75, and 77.

RAHMAN, S. M. Applying the TBC method in introductory programming courses. In: **2007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports**. [S.l.: s.n.], 2007. p. T1E–20–T1E–21. Citation on page 194.

RAHMAN, S. M.; JUELL, P. L. Applying Software Development Lifecycles in Teaching Introductory Programming Courses. In: **19th Conference on Software Engineering Education Training (CSEET'06)**. [S.l.: s.n.], 2006. p. 17–24. Citation on page 194.

RAJAGURU, D.; RAJESWARI, A.; BHUVANESHWARI, V.; VAGHEESAN, K. E-assessment of programming assignments in web service. In: **IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM -2012)**. [S.l.: s.n.], 2012. p. 484–489. Citation on page 195.

RAJALA, T.; KAILA, E.; LINDEN, R.; KURVINEN, E.; LOKKILA, E.; LAAKSO, M.-J.; SALAKOSKI, T. Automatically Assessed Electronic Exams in Programming Courses. In: **Proceedings of the Australasian Computer Science Week Multiconference**. New York, NY, USA: ACM, 2016. (ACSW '16), p. 11:1–11:8. ISBN 978-1-4503-4042-7. Available: <http://doi.acm.org/10.1145/2843043.2843062>. Citations on pages 94, 96, 98, and 197.

RANDOLPH, J. J. Findings from "a methodological review of the computer science education research: 2000–2005". **SIGCSE Bull.**, ACM, New York, NY, USA, v. 39, n. 4, p. 130–130, Dec. 2007. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1345375.1345434>. Citations on pages 21, 123, and 139.

RESNICK, M.; MALONEY, J.; MONROY-HERNANDEZ, A.; RUSK, N.; EASTMOND, E.; BRENNAN, K.; MILLNER, A.; ROSENBAUM, E.; SILVER, J.; SILVERMAN, B.; KAFAI, Y. Scratch: Programming for all. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 11, p. 60–67, Nov. 2009. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/1592761.1592779>. Citation on page 19.

REYNOLDS, L.; MAYO, Q.; ADAMO, D.; BRYCE, R. Improving Conceptual Understanding of Code with Bug Fixer. **J. Comput. Sci. Coll.**, v. 31, n. 2, p. 87–94, Dec. 2015. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2831432.2831445>. Citations on pages 87, 94, 130, and 197.

RICKEN, M.; CARTWRIGHT, R. Test-first Java Concurrency for the Classroom. In: **Proceedings of the 41st ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2010. (SIGCSE '10), p. 219–223. ISBN 978-1-4503-0006-3. Available: <http://doi.acm.org/10.1145/1734263.1734340>. Citation on page 198.

RING, B. A.; GIORDAN, J.; RANSBOTTOM, J. S. Problem Solving Through Programming: Motivating the Non-programmer. **J. Comput. Sci. Coll.**, v. 23, n. 3, p. 61–67, Jan. 2008. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=1295109.1295126>. Citation on page 192.

ROBERTS, G. H. B.; VERBYLA, J. L. M. An Online Programming Assessment Tool. In: **Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003. (ACE '03), p. 69–75. ISBN 0-909925-98-4. Available: <http://dl.acm.org/citation.cfm?id=858403.858412>. Citation on page 195.

ROBINS, A. Learning edge momentum: new account of outcomes in cs1. **Computer Science Education**, Taylor & Francis, v. 20, n. 1, p. 37–71, 2010. Citation on page 31.

____. The ongoing challenges of computer science education research. **Computer Science Education**, Routledge, v. 25, n. 2, p. 115–119, 2015. Available: <https://doi.org/10.1080/08993408.2015.1034350>. Citations on pages 21, 123, and 139.

ROBINS, A.; ROUNTREE, J.; ROUNTREE, N. Learning and teaching programming: A review and discussion. **Computer Science Education**, p. 137–172, 2003. Citations on pages 25, 31, and 32.

RODRIGUES, P. L. R.; FRANZ, L. P.; CHEIRAN, J. F. P.; SILVA, J. P. S. da; BORDIN, A. S. Coding dojo as a transforming practice in collaborative learning of programming: An experience report. In: **Proceedings of the 31st Brazilian Symposium on Software Engineering**. New York, NY, USA: ACM, 2017. (SBES'17), p. 348–357. ISBN 978-1-4503-5326-7. Available: <http://doi.acm.org/10.1145/3131151.3131180>. Citation on page 193.

ROMLI, R.; SULAIMAN, S.; ZAMLI, K. Z. Automatic programming assessment and test data generation a review on its approaches. In: **2010 International Symposium on Information Technology**. [S.l.: s.n.], 2010. v. 3, p. 1186–1192. Citation on page 197.

ROMLI, R.; SULAIMAN, S.; ZAMLI, K. Z. Current practices of programming assessment at higher learning institutions. **Communications in Computer and Information Science**, v. 179 CCIS, n. PART 1, p. 471–485, 2011. ISSN 18650929. Citation on page 194.

ROSIENE, J. A.; ROSIENE, C. P. Testing in the 'small'. **J. Comput. Sci. Coll.**, v. 19, n. 2, p. 314–318, Dec. 2003. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=948785.948829>. Citation on page 192.

ROSSLING, G.; HARTTE, S. WebTasks: Online Programming Exercises Made Easy. In: **Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2008. (ITiCSE '08), p. 363–363. ISBN 978-1-60558-078-4. Available: <http://doi.acm.org/10.1145/1384271.1384405>. Citation on page 197.

RUBIN, M. J. The Effectiveness of Live-coding to Teach Introductory Programming. In: **Proceeding of the 44th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2013. (SIGCSE '13), p. 651–656. ISBN 978-1-4503-1868-6. Available: <http://doi.acm.org/10.1145/2445196.2445388>. Citations on pages 94, 101, 103, 135, and 192.

RUBIO-SANCHEZ, M.; KINNUNEN, P.; PAREJA-FLORES, C.; VELAZQUEZ-ITURBIDE, A. Student perception and usage of an automated programming assessment tool. **Computers in Human Behavior**, v. 31, p. 453 – 460, 2014. ISSN 0747-5632. Available: <http://www.sciencedirect.com/science/article/pii/S0747563213001040>. Citations on pages 94, 99, and 196.

RUNESON, P. A survey of unit testing practices. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 4, p. 22–29, Jul. 2006. ISSN 0740-7459. Citation on page 64.

RUNESON, P.; HOST, M. Guidelines for conducting and reporting case study research in software engineering. **Empirical Software Engineering**, Springer US, v. 14, n. 2, p. 131–164, 2009. ISSN 1382-3256. Citation on page 41.

SAIKKONEN, R.; MALMI, L.; KORHONEN, A. Fully automatic assessment of programming exercises. In: **Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2001. (ITiCSE '01), p. 133–136. ISBN 1-58113-330-8. Available: <http://doi.acm.org/10.1145/377435.377666>. Citation on page 198.

SALLEH, N.; MENDES, E.; GRUNDY, J. Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review. **IEEE Transactions on Software Engineering**, v. 37, n. 4, p. 509–525, July 2011. ISSN 0098-5589. Citation on page 36.

____. Investigating the effects of personality traits on pair programming in a higher education setting through a family of experiments. **Empirical Softw. Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 3, p. 714–752, Jun. 2014. ISSN 1382-3256. Available: <http://dx.doi.org/10.1007/s10664-012-9238-4>. Citations on pages 20 and 22.

SANDERS, K.; AHMADZADEH, M.; CLEAR, T.; EDWARDS, S. H.; GOLDWEBER, M.; JOHNSON, C.; LISTER, R.; MCCARTNEY, R.; PATITSAS, E.; SPACCO, J. The canterbury questionbank: Building a repository of multiple-choice cs1 and cs2 questions. In: **Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports**. New York, NY, USA: ACM, 2013. (ITiCSE -WGR '13), p. 33–52. ISBN 978-1-4503-2665-0. Available: <http://doi.acm.org/10.1145/2543882.2543885>. Citation on page 199.

SANT, J. A. "mailing it in": Email-centric automated assessment. In: **Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2009. (ITiCSE '09), p. 308–312. ISBN 978-1-60558-381-5. Available: <http://doi.acm.org/10.1145/1562877.1562971>. Citation on page 195.

SAUVE, J. P.; NETO, O. L. A. Teaching Software Development with ATDD and Easyaccept. In: **Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2008. (SIGCSE '08), p. 542–546. ISBN 978-1-59593-799-5. Available: <http://doi.acm.org/10.1145/1352135.1352317>. Citations on pages 104 and 192.

SAUVÉ, J. P.; NETO, O. L. A.; CIRNE, W. Easyaccept: A tool to easily create, run and drive development with automated acceptance tests. In: **Proceedings of the 2006 International Workshop on Automation of Software Test**. New York, NY, USA: ACM, 2006. (AST '06), p. 111–117. ISBN 1-59593-408-1. Available: <http://doi.acm.org/10.1145/1138929.1138951>. Citations on pages 90, 97, 104, and 196.

SCATALON, L. P.; BARBOSA, E. F.; GARCIA, R. E. Challenges to integrate software testing into introductory programming courses. In: **2017 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2017. p. 1–9. Citation on page 191.

SCATALON, L. P.; CARVER, J. C.; GARCIA, R. E.; BARBOSA, E. F. Software testing in introductory programming courses: A systematic mapping study (accepted for publication). In: **50th ACM Technical Symposium on Computing Science Education (SIGCSE'19)**. Minneapolis, Minnesota, USA: [s.n.], 2019. Citation on page 75.

SCATALON, L. P.; FIORAVANTI, M. L.; PRATES, J. M.; GARCIA, R. E.; BARBOSA, E. F. A survey on graduates' curriculum-based knowledge gaps in software testing. In: **48th Annual Frontiers in Education Conference (FIE 2018)**. San Jose, California, EUA: [s.n.], 2018. Citation on page 63.

SCATALON, L. P.; PRATES, J. M.; SOUZA, D. M. de; BARBOSA, E. F.; GARCIA, R. E. Towards the role of test design in programming assignments. In: **30th IEEE Conference on Software Engineering Education and Training (CSEE&T 2017)**. Savannah, Georgia, EUA: [s.n.], 2017. Citations on pages 12, 109, and 136.

____. Towards the role of test design in programming assignments. In: **2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)**. [S.l.: s.n.], 2017. p. 170–179. Citations on pages 87, 90, 101, 129, and 194.

SCHAUB, S. Teaching CS1 with Web Applications and Test-driven Development. **SIGCSE Bull.**, v. 41, n. 2, p. 113–117, Jun. 2009. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1595453.1595487>. Citation on page 193.

SCHULTE, C.; BENNEDSEN, J. What do teachers teach in introductory programming? In: **Proceedings of the Second International Workshop on Computing Education Research (ICER '06)**. New York, NY, USA: ACM, 2006. p. 17–28. ISBN 1-59593-494-4. Citation on page 28.

SHAFFER, C. A.; AKBAR, M.; ALON, A. J. D.; STEWART, M.; EDWARDS, S. H. Getting algorithm visualizations into the classroom. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)**. New York, NY, USA: ACM, 2011. p. 129–134. ISBN 978-1-4503-0500-6. Citation on page 37.

SHAFFER, C. A.; COOPER, M. L.; ALON, A. J. D.; AKBAR, M.; STEWART, M.; PONCE, S.; EDWARDS, S. H. Algorithm visualization: The state of the field. **ACM Transactions on Computing Education**, ACM, New York, NY, USA, v. 10, n. 3, p. 9:1–9:22, Aug. 2010. ISSN 1946-6226. Citation on page 37.

SHAFFER, S. C. Ludwig: An online programming tutoring and assessment system. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 37, n. 2, p. 56–60, Jun. 2005. ISSN 0097-8418. Available: <http://doi.acm.org/10.1145/1083431.1083464>. Citation on page 198.

SHAMS, Z. 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**. New York, NY, USA: ACM, 2013. (SPLASH '13), p. 37–40. ISBN 978-1-4503-1995-9. Available: <http://doi.acm.org/10.1145/2508075.2508078>. Citations on pages 91, 101, 134, and 196.

____. Automatically Assessing the Quality of Student-written Tests. In: **Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research**. New York, NY, USA: ACM, 2013. (ICER '13), p. 189–190. ISBN 978-1-4503-2243-0. Available: <http://doi.acm.org/10.1145/2493394.2493428>. Citation on page 199.

SHAMS, Z.; EDWARDS, S. H. 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**. New York, NY, USA: ACM, 2013. (ICER '13), p. 53–58. ISBN 978-1-4503-2243-0. Available: <http://doi.acm.org/10.1145/2493394.2493402>. Citations on pages 91, 102, 134, and 196.

____. 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**. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 534–539. ISBN 978-1-4503-2966-8. Available: <http://doi.acm.org/10.1145/2676723.2677300>. Citations on pages 94, 131, and 199.

SHEARD, J.; SIMON, S.; HAMILTON, M.; LONNBERG, J. Analysis of research into the teaching and learning of programming. In: **Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09)**. New York, NY, USA: ACM, 2009. p. 93–104. ISBN 978-1-60558-615-1. Citations on pages 19, 21, 22, 25, 123, and 139.

SHETH, S.; BELL, J.; KAISER, G. Halo (highly addictive, socially optimized) software engineering. In: **1st International Workshop on Games and Software Engineering (GAS'11)**. [s.n.], 2011. Available: <https://pdfs.semanticscholar.org/2aa5/56e8dce6b5d1aa91279da52840f7587559b1.pdf>. Citation on page 198.

SHULL, F.; CRUZES, D.; BASILI, V.; MENDONCA, M. Simulating families of studies to build confidence in defect hypotheses. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 47, n. 15, p. 1019–1032, Dec. 2005. ISSN 0950-5849. Citations on pages 55 and 135.

SHULL, F.; SINGER, J.; SJOBERG, D. I. **Guide to Advanced Empirical Software Engineering**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN 184800043X. Citation on page 40.

SHULL, F. J.; CARVER, J. C.; VEGAS, S.; JURISTO, N. The role of replications in empirical software engineering. **Empirical Software Engineering**, Kluwer Academic Publishers, v. 13, n. 2, p. 211–218, 2008. Citation on page 42.

SIOSON, A. Experiences on the use of an automatic C++ solution grader system. In: **IISA 2013 - 4th International Conference on Information, Intelligence, Systems and Applications**. [S.l.: s.n.], 2013. p. 18–21. ISBN 978-1-4799-0771-7. Citation on page 196.

SJOBERG, D.; ANDA, B.; ARISHOLM, E.; DYBA, T.; JORGENSEN, M.; KARAHASANOVI, A.; VOKA, M. Challenges and recommendations when increasing the realism of controlled software engineering experiments. In: CONRADI, R.; WANG, A. (Ed.). **Empirical Methods and Studies in Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science, v. 2765). p. 24–38. ISBN 978-3-540-40672-3. Citation on page 45.

SJOBERG, D. I. K.; DYBA, T.; JORGENSEN, M. The future of empirical methods in software engineering research. In: **Future of Software Engineering (FOSE '07)**. Washington, DC, USA: IEEE Computer Society, 2007. p. 358–378. ISBN 0-7695-2829-5. Citation on page 40.

SMITH, J.; TESSLER, J.; KRAMER, E.; LIN, C. Using Peer Review to Teach Software Testing. In: **Proceedings of the Ninth Annual International Conference on International Computing Education Research**. New York, NY, USA: ACM, 2012. (ICER '12), p. 93–98.

ISBN 978-1-4503-1604-0. Available: <http://doi.acm.org/10.1145/2361276.2361295>. Citation on page 193.

SMITH, R.; TANG, T.; WARREN, J.; RIXNER, S. An automated system for interactively learning software testing. In: **Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2017. (ITiCSE '17), p. 98–103. ISBN 978-1-4503-4704-4. Available: <http://doi.acm.org/10.1145/3059009.3059022>. Citations on pages 87, 94, and 197.

SMITH, S.; STOECKLIN, S. What We Can Learn from Extreme Programming. **J. Comput. Sci. Coll.**, v. 17, n. 2, p. 144–151, Dec. 2001. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=775339.775368>. Citation on page 194.

SNYDER, J.; EDWARDS, S. H.; PEREZ-QUINONES, M. A. LIFT: Taking GUI Unit Testing to New Heights. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 643–648. ISBN 978-1-4503-0500-6. Available: <http://doi.acm.org/10.1145/1953163.1953343>. Citation on page 198.

SNYDER, R. M. Teacher Specification and Student Implementation of a Unit Testing Methodology in an Introductory Programming Course. **J. Comput. Sci. Coll.**, v. 19, n. 3, p. 22–32, Jan. 2004. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=948835.948837>. Citation on page 193.

SOLARI, M.; VEGAS, S. Classifying and analysing replication packages for software engineering experimentation. In: **7th International Conference on Product Focused Software Process Improvement (PROFES 2006)-Workshop Series in Empirical Software Engineering (WSESE)**. [S.l.: s.n.], 2006. Citation on page 55.

SOLOWAY, E.; BONAR, J.; EHRLICH, K. Cognitive strategies and looping constructs: An empirical study. **Commun. ACM**, ACM, New York, NY, USA, v. 26, n. 11, p. 853–860, Nov. 1983. ISSN 0001-0782. Available: <http://doi.acm.org/10.1145/182.358436>. Citation on page 31.

SOMMERVILLE, I. **An embedded control system for a personal insulin pump**. 2010. Available at <http://iansommerville.com/software-engineering-book/case-studies/a-personal-insulin-pump/>. Citation on page 119.

SORVA, J.; KARAVIRTA, V.; MALMI, L. A review of generic program visualization systems for introductory programming education. **ACM Transactions on Computing Education**, ACM, New York, NY, USA, v. 13, n. 4, p. 15:1–15:64, Nov. 2013. ISSN 1946-6226. Citation on page 37.

SOUZA, D. M. d.; MALDONADO, J. C.; BARBOSA, E. F. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In: **2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)**. [S.l.: s.n.], 2011. p. 1–10. Citations on pages 93, 98, and 196.

SOUZA, D. M. d.; OLIVEIRA, B. H.; MALDONADO, J. C.; SOUZA, S. R. S.; BARBOSA, E. F. Towards the use of an automatic assessment system in the teaching of software testing. In: **2014 IEEE Frontiers in Education Conference (FIE) Proceedings**. [S.l.: s.n.], 2014. p. 1–8. Citations on pages 93, 98, and 196.

SOUZA, D. M. de; ISOTANI, S.; BARBOSA, E. F. Teaching novice programmers using progtest. **International Journal of Knowledge and Learning**, 2015. Available: <http://www.producao.usp.br/bitstream/handle/BDPI/51271/2744279.pdf?sequence=1&isAllowed=y>. Citations on pages 90, 91, 93, 94, 98, 100, and 197.

SOUZA, D. M. de; KOLLING, M.; BARBOSA, E. F. Most common fixes students use to improve the correctness of their programs. In: **2017 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2017. p. 1–9. Citations on pages 98, 100, 133, and 199.

SPACCO, J.; FOSSATI, D.; STAMPER, J.; RIVERS, K. Towards Improving Programming Habits to Create Better Computer Science Course Outcomes. In: **Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2013. (ITiCSE '13), p. 243–248. ISBN 978-1-4503-2078-8. Available: <http://doi.acm.org/10.1145/2462476.2465594>. Citations on pages 36, 94, 96, 98, 99, and 200.

SPACCO, J.; HOVEMEYER, D.; PUGH, W. An Eclipse-based Course Project Snapshot and Submission System. In: **Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange**. New York, NY, USA: ACM, 2004. (eclipse '04), p. 52–56. Available: <http://doi.acm.org/10.1145/1066129.1066140>. Citation on page 197.

SPACCO, J.; HOVEMEYER, D.; PUGH, W.; EMAD, F.; HOLLINGSWORTH, J. K.; PADUA-PEREZ, N. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In: **Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2006. (ITICSE '06), p. 13–17. ISBN 1-59593-055-8. Available: <http://doi.acm.org/10.1145/1140124.1140131>. Citation on page 198.

SPACCO, J.; PUGH, W. Helping Students Appreciate Test-driven Development (TDD). In: **Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications**. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 907–913. ISBN 1-59593-491-X. Available: <http://doi.acm.org/10.1145/1176617.1176743>. Citations on pages 92, 101, 134, and 200.

SPACCO, J.; PUGH, W.; AYEWAH, N.; HOVEMEYER, D. 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**. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 669–670. ISBN 1-59593-491-X. Available: <http://doi.acm.org/10.1145/1176617.1176665>. Citation on page 197.

SPACCO, J.; STRECKER, J.; HOVEMEYER, D.; PUGH, W. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. In: **Proceedings of the 2005 International Workshop on Mining Software Repositories**. New York, NY, USA: ACM, 2005. (MSR '05), p. 1–5. ISBN 1-59593-123-6. Available: <http://doi.acm.org/10.1145/1082983.1083149>. Citations on pages 99 and 196.

SRIDHARA, S.; HOU, B.; LU, J.; DENERO, J. Fuzz Testing Projects in Massive Courses. In: **Proceedings of the Third (2016) ACM Conference on Learning Scale**. New York, NY, USA: ACM, 2016. (L@S '16), p. 361–367. ISBN 978-1-4503-3726-7. Available: <http://doi.acm.org/10.1145/2876034.2876050>. Citations on pages 98, 99, and 197.

STEINBERG, D. H. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. In: **Proceedings of XP Universe Conference**. [s.n.],

2001. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.5827>. Citation on page 199.

STRIEWE, M.; GOEDICKE, M. Using run time traces in automated programming tutoring. In: **Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2011. (ITiCSE '11), p. 303–307. ISBN 978-1-4503-0697-3. Available: <http://doi.acm.org/10.1145/1999747.1999832>. Citation on page 195.

SULEMAN, H. Automatic Marking with Sakai. In: **Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology**. New York, NY, USA: ACM, 2008. (SAICSIT '08), p. 229–236. ISBN 978-1-60558-286-3. Available: <http://doi.acm.org/10.1145/1456659.1456686>. Citation on page 198.

SUMMET, J.; KUMAR, D.; O'HARA, K.; WALKER, D.; NI, L.; BLANK, D.; BALCH, T. Personalizing cs1 with robots. In: **Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)**. New York, NY, USA: ACM, 2009. p. 433–437. ISBN 978-1-60558-183-5. Citation on page 38.

SUN, Y.; JONES, E. L. Specification-driven Automated Testing of GUI-based Java Programs. In: **Proceedings of the 42Nd Annual Southeast Regional Conference**. New York, NY, USA: ACM, 2004. (ACM-SE 42), p. 140–145. ISBN 1-58113-870-9. Available: <http://doi.acm.org/10.1145/986537.986570>. Citation on page 197.

SURAKKA, S. What subjects and skills are important for software developers? **Communications of the ACM**, ACM, New York, NY, USA, v. 50, n. 1, p. 73–78, 2007. ISSN 0001-0782. Citation on page 19.

TANG, T.; SMITH, R.; RIXNER, S.; WARREN, J. Data-Driven Test Case Generation for Automated Programming Assessment. In: **Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2016. (ITiCSE '16), p. 260–265. ISBN 978-1-4503-4231-5. Available: <http://doi.acm.org/10.1145/2899415.2899423>. Citations on pages 94, 99, and 197.

TEUSNER, R.; HILLE, T.; HAGEDORN, C. Aspects on finding the optimal practical programming exercise for moocs. In: **2017 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2017. p. 1–8. Citations on pages 94, 101, 102, 134, and 194.

TEW, A. E.; GUZDIAL, M. The fcs1: A language independent assessment of cs1 knowledge. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 111–116. ISBN 978-1-4503-0500-6. Available: <http://doi.acm.org/10.1145/1953163.1953200>. Citations on pages 20 and 34.

TEW, A. E.; MCCRACKEN, W. M.; GUZDIAL, M. Impact of alternative introductory courses on programming concept understanding. In: **Proceedings of the First International Workshop on Computing Education Research (ICER '05)**. New York, NY, USA: [s.n.], 2005. p. 25–35. ISBN 1-59593-043-4. Citation on page 35.

THOMPSON, E.; HUNT, L.; KINSHUK, K. Exploring Learner Conceptions of Programming. In: **Proceedings of the 8th Australasian Conference on Computing Education - Volume 52**.

Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006. (ACE '06), p. 205–211. ISBN 1-920682-34-1. Available: <http://dl.acm.org/citation.cfm?id=1151869.1151896>. Citation on page 200.

THORNTON, M.; EDWARDS, S. H.; TAN, R. P.; PEREZ-QUINONES, M. A. Supporting Student-written Tests of Gui Programs. In: **Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2008. (SIGCSE '08), p. 537–541. ISBN 978-1-59593-799-5. Available: <http://doi.acm.org/10.1145/1352135.1352316>. Citations on pages 87, 95, 97, 98, 130, 134, and 196.

THURNER, V.; BOTTCHER, A. An objects first, tests second approach for software engineering education. In: **2015 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2015. p. 1–5. Citations on pages 109 and 192.

TIANTIAN, W.; XIAOHONG, S.; PEIJUN, M.; YUYING, W.; KUANQUAN, W. AutoLEP: An Automated Learning and Examination System for Programming and its Application in Programming Course. In: **2009 First International Workshop on Education Technology and Computer Science**. [S.l.: s.n.], 2009. v. 1, p. 43–46. Citation on page 195.

TILLMANN, N.; HALLEUX, J. d.; XIE, T.; BISHOP, J. Pex4fun: A web-based environment for educational gaming via automated test generation. In: **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2013. p. 730–733. Citation on page 198.

TILLMANN, N.; HALLEUX, J. de; XIE, T.; BISHOP, J. Constructing Coding Duels in Pex4fun and Code Hunt. In: **Proceedings of the 2014 International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2014. (ISSTA 2014), p. 445–448. ISBN 978-1-4503-2645-2. Available: <http://doi.acm.org/10.1145/2610384.2628054>. Citation on page 196.

TREMBLAY, G.; GUERIN, F.; PONS, A.; SALAH, A. Oto, a generic and extensible tool for marking programming assignments. **Software: Practice and Experience**, John Wiley & Sons, Ltd., v. 38, n. 3, p. 307–333, 2008. ISSN 1097-024X. Available: <http://dx.doi.org/10.1002/spe.839>. Citation on page 195.

TREMBLAY, G.; LAFOREST, L.; SALAH, A. Extending a Marking Tool with Simple Support for Testing. In: **Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2007. (ITiCSE '07), p. 313–313. ISBN 978-1-59593-610-3. Available: <http://doi.acm.org/10.1145/1268784.1268879>. Citation on page 197.

TREMBLAY, G.; LESSARD, P. A marking language for the oto assignment marking tool. In: **Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2011. (ITiCSE '11), p. 148–152. ISBN 978-1-4503-0697-3. Available: <http://doi.acm.org/10.1145/1999747.1999791>. Citation on page 195.

TURNER, S. A. Looking Glass: A C++ Library for Testing Student Programs Through Reflection. In: **Proceedings of the 46th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2015. (SIGCSE '15), p. 528–533. ISBN 978-1-4503-2966-8. Available: <http://doi.acm.org/10.1145/2676723.2677281>. Citation on page 198.

UREEL, L. C.; WALLACE, C. WebTA: Automated iterative critique of student programming assignments. In: **2015 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2015. p. 1–9. Citation on page 196.

UTTING, I.; TEW, A. E.; MCCRACKEN, M.; THOMAS, L.; BOUVIER, D.; FRYE, R.; PATERSON, J.; CASPERSEN, M.; KOLIKANT, Y. B.-D.; SORVA, J.; WILUSZ, T. A fresh look at novice programmers' performance and their teachers' expectations. In: **Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports**. New York, NY, USA: ACM, 2013. (ITiCSE -WGR '13), p. 15–32. ISBN 978-1-4503-2665-0. Available: <http://doi.acm.org/10.1145/2543882.2543884>. Citations on pages 13, 20, 22, 25, 32, 34, 38, and 111.

____. A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations. In: **Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-working Group Reports**. New York, NY, USA: ACM, 2013. (ITiCSE -WGR '13), p. 15–32. ISBN 978-1-4503-2665-0. Available: <http://doi.acm.org/10.1145/2543882.2543884>. Citations on pages 90, 94, 104, 133, 134, and 199.

VALENTINE, D. W. Cs educational research: A meta-analysis of sigcse technical symposium proceedings. In: **Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2004. (SIGCSE '04), p. 255–259. ISBN 1-58113-798-2. Available: <http://doi.acm.org/10.1145/971300.971391>. Citations on pages 19, 21, 123, and 139.

VALLE, P. H. D.; TODA, A. M.; BARBOSA, E. F.; MALDONADO, J. C. Educational games: A contribution to software testing education. In: **2017 IEEE Frontiers in Education Conference (FIE)**. [S.l.: s.n.], 2017. p. 1–8. Citation on page 199.

VANDEGRIFT, T.; CARUSO, T.; HILL, N.; SIMON, B. Experience Report: Getting Novice Programmers to THINK About Improving Their Software Development Process. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 493–498. ISBN 978-1-4503-0500-6. Available: <http://doi.acm.org/10.1145/1953163.1953307>. Citation on page 195.

VEE, A. Understanding computer programming as a literacy. **Literacy in Composition Studies**, v. 1, n. 2, 2013. Citation on page 19.

VEGAS, S.; JURISTO, N.; MORENO, A.; SOLARI, M.; LETELIER, P. Analysis of the influence of communication between researchers on experiment replication. In: **Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering**. New York, NY, USA: ACM, 2006. p. 28–37. ISBN 1-59593-218-6. Citation on page 54.

VENABLES, A.; HAYWOOD, L. Programming students need instant feedback! In: **Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003. (ACE '03), p. 267–272. ISBN 0-909925-98-4. Available: <http://dl.acm.org/citation.cfm?id=858403.858436>. Citation on page 195.

VIHAVAINEN, A.; AIRAKSINEN, J.; WATSON, C. 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)**. New York, NY, USA: ACM, 2014. p. 19–26. ISBN 978-1-4503-2755-8. Citations on pages 20 and 35.

VIHAVAINEN, A.; VIKBERG, T.; LUUKKAINEN, M.; PARTEL, M. Scaffolding Students' Learning Using Test My Code. In: **Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2013. (ITiCSE '13), p. 117–122. ISBN 978-1-4503-2078-8. Available: <http://doi.acm.org/10.1145/2462476.2462501>. Citation on page 196.

VINCENZI, A.; DELAMARO, M.; HOHN, E.; MALDONADO, J. C. Testing techniques in software engineering. In: ____. [S.l.]: Springer, 2010. chap. Functional, Control and Data Flow, and Mutation Testing: Theory and Practice. Citation on page 119.

VUJOSEVIC-JANICIC, M.; NIKOLIC, M.; TOSIC, D.; KUNCAK, V. Software verification and graph similarity for automated evaluation of students' assignments. **Information and Software Technology**, v. 55, n. 6, p. 1004 – 1016, 2013. ISSN 0950-5849. Available: <http://www.sciencedirect.com/science/article/pii/S0950584912002406>. Citations on pages 87, 88, 89, 99, 130, and 196.

WANG, T.; SU, X.; MA, P.; WANG, Y.; WANG, K. Ability-training-oriented automated assessment in introductory programming course. **Computers & Education**, v. 56, n. 1, p. 220 – 226, 2011. ISSN 0360-1315. Serious Games. Available: <http://www.sciencedirect.com/science/article/pii/S0360131510002241>. Citations on pages 87, 90, 98, 130, and 196.

WATSON, C.; LI, F. W. Failure rates in introductory programming revisited. In: **Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)**. New York, NY, USA: ACM, 2014. p. 39–44. ISBN 978-1-4503-2833-3. Citations on pages 20 and 31.

WELLINGTON, C. A.; BRIGGS, T. H.; GIRARD, C. D. Experiences Using Automated 4ests and 4est Driven Development in Computer 9cience I. In: **Agile 2007 (AGILE 2007)**. [S.l.: s.n.], 2007. p. 106–112. Citation on page 192.

WHALLEY, J.; KASTO, N. How difficult are novice code writing tasks?: A software metrics approach. In: **Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2014. (ACE '14), p. 105–112. ISBN 978-1-921770-31-9. Available: <http://dl.acm.org/citation.cfm?id=2667490.2667503>. Citations on pages 88, 89, 101, 102, and 199.

WHALLEY, J. L.; PHILPOTT, A. A Unit Testing Approach to Building Novice Programmers' Skills and Confidence. In: **Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114**. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2011. (ACE '11), p. 113–118. ISBN 978-1-920682-94-1. Available: <http://dl.acm.org/citation.cfm?id=2459936.2459950>. Citations on pages 20, 74, 75, 140, and 193.

WICK, M.; STEVENSON, D.; WAGNER, P. Using Testing and JUnit Across the Curriculum. In: **Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2005. (SIGCSE '05), p. 236–240. ISBN 1-58113-997-7. Available: <http://doi.acm.org/10.1145/1047344.1047427>. Citations on pages 109 and 191.

WILCOX, C. Testing Strategies for the Automated Grading of Student Programs. In: **Proceedings of the 47th ACM Technical Symposium on Computing Science Education**. New York, NY, USA: ACM, 2016. (SIGCSE '16), p. 437–442. ISBN 978-1-4503-3685-7. Available: <http://doi.acm.org/10.1145/2839509.2844616>. Citation on page 198.

WILLIAMS, L.; KESSLER, R. R.; CUNNINGHAM, W.; JEFFRIES, R. Strengthening the case for pair programming. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 17, n. 4, p. 19–25, Jul. 2000. ISSN 0740-7459. Citations on pages 36 and 59.

WILLIAMS, L.; KREBS, W.; LAYMAN, L.; ANTON, A.; ABRAHAMSSON, P. Toward a framework for evaluating extreme programming. In: **Assessment in Software Engineering (EASE)**. [S.l.: s.n.], 2004. Citations on pages 13, 23, 55, 60, 61, 62, and 124.

WILLIAMS, L.; LAYMAN, L.; ABRAHAMSSON, P. On establishing the essential components of a technology-dependent framework: A strawman framework for industrial case study-based research. In: **Proceedings of the 2005 Workshop on Realising Evidence-based Software Engineering**. New York, NY, USA: ACM, 2005. (REBSE '05), p. 1–5. ISBN 1-59593-121-X. Available: <http://doi.acm.org/10.1145/1082983.1083179>. Citations on pages 55 and 60.

WILLIAMS, L. A. **The Collaborative Software Process**. Phd Thesis (PhD Thesis) — University of Utah, 2000. Citation on page 59.

WILSON, C. Hour of code—a record year for computer science. **ACM Inroads**, ACM, New York, NY, USA, v. 6, n. 1, p. 22–22, Feb. 2015. ISSN 2153-2184. Available: <http://doi.acm.org/10.1145/2723168>. Citation on page 19.

WOHLIN, C. 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**. New York, NY, USA: ACM, 2014. (EASE'14), p. 38:1–38:10. ISBN 978-1-4503-2476-2. Citations on pages 23 and 77.

WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M.; REGNELL, B.; WESSLEN, A. **Experimentation in Software Engineering: An introduction**. 1. ed. Boston, USA: Kluwer Academic Publishers, 2000. Citation on page 40.

____. **Experimentation in Software Engineering**. 2. ed. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435. Citations on pages 11, 13, 23, 42, 43, 44, 45, 51, 52, 53, 54, 56, 60, 110, and 113.

XIE, T.; BISHOP, J.; HORSPOOL, R. N.; TILLMANN, N.; HALLEUX, J. d. Crowdsourcing Code and Process via Code Hunt. In: **2015 IEEE/ACM 2nd International Workshop on CrowdSourcing in Software Engineering**. [S.l.: s.n.], 2015. p. 15–16. Citation on page 198.

YI, J.; AHMED, U. Z.; KARKARE, A.; TAN, S. H.; ROYCHOUDHURY, A. A feasibility study of using automated program repair for introductory programming assignments. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 740–751. ISBN 978-1-4503-5105-8. Available: <http://doi.acm.org/10.1145/3106237.3106262>. Citations on pages 100 and 197.

ZANDEN, B. V.; ANDERSON, D.; TAYLOR, C.; DAVIS, W.; BERRY, M. W. Codeassessor: An Interactive, Web-based Tool for Introductory Programming. **J. Comput. Sci. Coll.**, v. 28, n. 2, p. 73–80, Dec. 2012. ISSN 1937-4771. Available: <http://dl.acm.org/citation.cfm?id=2382887.2382900>. Citation on page 195.

ZELLER, A. Making Students Read and Review Code. In: **Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2000. (ITiCSE '00), p. 89–92. ISBN 1-58113-207-7. Available: <http://doi.acm.org/10.1145/343048.343090>. Citation on page 195.

ZHANG, H.; BABAR, M. A.; TELL, P. Identifying relevant studies in software engineering. **Information and Software Technology**, v. 53, n. 6, p. 625 – 637, 2011. ISSN 0950-5849. Special Section: Best papers from the APSEC. Citations on pages 23 and 77.

ZHU, G.; CHEN, Y. Knowledge-based links for automatic interaction with programming online judges. **Journal of Software**, v. 8, n. 5, p. 1209–1218, 2013. ISSN 1796217X. Citation on page 198.

ZIMMERMAN, D. M.; KINIRY, J. R.; FAIRMICHAEL, F. Toward instant gradeification. In: **2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)**. [S.l.: s.n.], 2011. p. 406–410. Citation on page 198.

ZINGARO, D. Examining interest and grades in computer science 1: A study of pedagogy and achievement goals. **Trans. Comput. Educ.**, ACM, New York, NY, USA, v. 15, n. 3, p. 14:1–14:18, Jul. 2015. ISSN 1946-6226. Available: <http://doi.acm.org/10.1145/2802752>. Citation on page 20.

ZINGARO, D.; CHERENKOVA, Y.; KARPOVA, O.; PETERSEN, A. Facilitating Code-writing in PI Classes. In: **Proceeding of the 44th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2013. (SIGCSE '13), p. 585–590. ISBN 978-1-4503-1868-6. Available: <http://doi.acm.org/10.1145/2445196.2445369>. Citation on page 198.

ZINGARO, D.; PORTER, L. Peer instruction in computing: The value of instructor intervention. **Computers & Education**, Elsevier Science Ltd., Oxford, UK, UK, v. 71, p. 87–96, Feb. 2014. ISSN 0360-1315. Citation on page 35.

APPENDIX

A

SURVEY QUESTIONNAIRE

1. What is your current position in the company?

2. How many years of work experience do you have in software development?

3. What is your university degree in?

   ☐ computer science

   ☐ computer engineering

   ☐ information systems

   ☐ software engineering

   ☐ other:

4. What computing courses addressed software testing in your major?

   ☐ introductory programming courses

   ☐ software engineering course

   ☐ software testing course

   ☐ extracurricular short course

   ☐ other:

5. Which programming languages are generally used in your projects?

   ☐ Java

   ☐ C

   ☐ C++

☐ Python

☐ C#

☐ PHP

☐ JavaScript

☐ Ruby

☐ Perl

☐ Swift

☐ other:

The remainder of the questionnaire addresses elements that characterize the testing activity. In questions 6 to 10, for each element, please analyze the testing approaches listed in each line and check the following columns:

- Applied in industry, if you have applied the testing approach in your job.
- Concept addressed in major, if you have learned about the theory of this testing approach during your major.
- Practice activities in major, if you have completed hands-on activities (such as programming/testing assignments) that gave you the opportunity to put the testing approach into practice.

6. Types of systems under test

	Applied in industry	Concepts addressed in major	Practice activities in major
Web applications	☐	☐	☐
Mobile applications	☐	☐	☐
Object oriented software	☐	☐	☐
Aspect oriented software	☐	☐	☐
Concurrent programs	☐	☐	☐

Other types of systems under test:

7. Testing levels

	Applied in industry	Concepts addressed in major	Practice activities in major
Unit testing	☐	☐	☐
Integration testing	☐	☐	☐
System testing	☐	☐	☐
Regression testing	☐	☐	☐

Other testing levels:

	Applied in industry	Concepts addressed in major	Practice activities in major
Functionality testing	☐	☐	☐
Performance testing	☐	☐	☐
GUI testing	☐	☐	☐
Usability testing	☐	☐	☐
Security testing	☐	☐	☐
User acceptance testing	☐	☐	☐

8. Test types

   Other test types:

9. Testing approach in the development process

	Applied in industry	Concepts addressed in major	Practice activities in major
Test-driven (first) development (TDD)	☐	☐	☐
Test-last development	☐	☐	☐

   Other testing approaches:

10. Test case generation techniques

	Applied in industry	Concepts addressed in major	Practice activities in major
Client requirements/ user stories	☐	☐	☐
Category partitioning	☐	☐	☐
Boundary value analysis	☐	☐	☐
Cause-effect graph	☐	☐	☐
Finite state machine	☐	☐	☐
Control flow graph	☐	☐	☐
Data flow analysis	☐	☐	☐
Mutation analysis	☐	☐	☐

   Other test techniques:

11. Which testing tools/frameworks do you use in your company?

12. If you wish, please comment other aspects about your experience with learning software testing in your major and industry testing practices.

APPENDIX B

MAPPING RESULTS

The following sections list all selected papers in the systematic mapping. Each section contains the papers mapped to an identified topic. For each selected paper, we indicate the study ID (in the format S#), the publication year, the publication venue and the evaluation method used in the study.

B.1 Curriculum

The topic *curriculum* includes the papers listed in Table **??**, which discuss the integration of testing in the computing curriculum as a whole or in individual programming courses.

Table 38 – Selected papers mapped to topic "curriculum"

study ID	**reference**	**year**	**venue name**	**evaluation method**
S1	Edwards (2003b)	2003	OOPSLA	descriptive
S2	Adams (2009)	2009	SIGCSE	descriptive
S3	Cowling (2012)	2012	ICSE	descriptive
S4	Frezza (2002)	2002	FIE	experience report
S5	Christensen (2003)	2003	ITiCSE	experience report
S6	Leska (2004)	2004	Journal of Computing in Small Colleges	experience report
S7	Wick, Stevenson and Wagner (2005)	2005	SIGCSE	experience report
S8	Dorin (2007)	2007	SIGCSE Bulletin	experience report
S9	Gestwicki (2018)	2018	SIGCSE	experience report
S10	Scatalon, Barbosa and Garcia (2017)	2017	FIE	literature review
S11	Jones (2000b)	2000	Australasian Conference on Computing Education	not applicable
S12	Jones (2001)	2001	SIGCSE	not applicable
S13	Jones (2001)	2001	FIE	not applicable
S14	Heliotis and Zanibbi (2011)	2011	Journal of Computing in Small Colleges	survey

B.2 Teaching methods

The topic *teaching methods* includes the papers listed in Table 39, which investigate methods to teach programming with the integration of software testing.

Table 39 – Selected papers mapped to topic "teaching methods"

study ID	**reference**	**year**	**venue name**	**evaluation method**
S15	Olan (2003)	2003	Journal of Computing in Small Colleges	descriptive
S16	Goldwasser (2002)	2002	SIGCSE	experience report
S17	Edwards (2003c)	2003	OOPSLA	experience report
S18	Kolling *et al.* (2003)	2003	Computer Science Education	experience report
S19	Miller (2004)	2004	Journal of Computing in Small Colleges	experience report
S20	Leska and Rabung (2005)	2005	Journal of Computing in Small Colleges	experience report
S21	Edwards and Perez-Quinones (2007)	2007	Journal of Computing in Small Colleges	experience report
S22	Wellington, Briggs and Girard (2007)	2007	Agile	experience report
S23	Carlson (2008)	2008	Agile	experience report
S24	Gotel, Scharff and Wildenberg (2008)	2008	SIGCSE Bulletin	experience report
S25	Ring, Giordan and Ransbottom (2008)	2008	Journal of Computing in Small Colleges	experience report
S26	Sauve and Neto (2008)	2008	SIGCSE	experience report
S27	Proulx (2009)	2009	SIGCSE	experience report
S28	Kart (2013)	2013	Journal of Computing in Small Colleges	experience report
S29	Gonzalez-Guerra and Leal-Flores (2014)	2014	ICCSE	experience report
S30	Thurner and Bottcher (2015)	2015	FIE	experience report
S31	Joshi and Desai (2016)	2016	T4E	experience report
S32	Barriocanal *et al.* (2002)	2002	SIGCSE Bulletin	experimental
S33	Edwards (2003a)	2003	Journal on Educational Resources in Computing	experimental
S34	Edwards (2004)	2004	SIGCSE	experimental
S35	Janzen and Saiedian (2006b)	2006	SIGCSE	experimental
S36	Janzen and Saiedian (2008)	2008	SIGCSE	experimental
S37	Hilton and Janzen (2012)	2012	ITiCSE	experimental
S38	Isomottonen and Lappalainen (2012)	2012	ACM Inroads	experimental
S39	Rubin (2013)	2013	SIGCSE	experimental
S40	Politz, Krishnamurthi and Fisler (2014)	2014	ICER	experimental
S41	Oliveira *et al.* (2015)	2015	FIE	experimental
S42	Li and Morreale (2016)	2016	Journal of Computing in Small Colleges	experimental
S43	Politz *et al.* (2016)	2016	SIGCSE	experimental
S44	Lee, Marepalli and Yang (2017)	2017	Journal of Computing in Small Colleges	experimental
S45	Matthies, Treffer and Uflacker (2017)	2017	FIE	experimental
S46	Jones (2000c)	2000	ADMI	not applicable
S47	Rosiene and Rosiene (2003)	2003	Journal of Computing in Small Colleges	not applicable

(continued in the next page)

Selected papers mapped to topic “teaching methods”(continued)

study ID	reference	year	venue name	evaluation method
S48	Snyder (2004)	2004	Journal of Computing in Small Colleges	not applicable
S49	Girard and Wellington (2006)	2006	FIE	not applicable
S50	Allison (2007)	2007	Journal of Computing in Small Colleges	not applicable
S51	Briggs and Girard (2007)	2007	Journal of Computing in Small Colleges	not applicable
S52	Gaspar and Langevin (2007b)	2007	SIGITE	not applicable
S53	Gaspar and Langevin (2007a)	2007	EISTA	not applicable
S54	Hernan-Losada, Pareja-Flores and Velazquez-Iturbide (2008)	2008	ICALT	not applicable
S55	Schaub (2009)	2009	SIGCSE Bulletin	not applicable
S56	Proulx (2011)	2011	book	not applicable
S57	Beaubouef and Zhang (2012)	2012	Journal of Computing in Small Colleges	not applicable
S58	Brannock and Napier (2012)	2012	Conference on Information Technology Education	not applicable
S59	Horvath (2012)	2012	ICETA	not applicable
S60	Alkadi and Alkadi (2002)	2002	IEEE Aerospace Conference	survey
S61	Barbosa *et al.* (2003)	2003	CSEE&T	survey
S62	Whalley and Philpott (2011)	2011	Australasian Computing Education Conference	survey
S63	Smith *et al.* (2012)	2012	ICER	survey
S64	Chen and Hall (2013)	2013	ITiCSE	survey
S65	Gaspar *et al.* (2013)	2013	SIGITE	survey
S66	Basu *et al.* (2015)	2015	Learning Scale Conference	survey
S67	Rodrigues *et al.* (2017)	2017	SBES	survey

B.3 Course materials

The topic *course materials* includes the papers listed in Table ??, which investigate how to incorporate testing concepts into course materials of introductory courses.

Table 40 – Selected papers mapped to topic “course materials”

study ID	reference	year	venue name	evaluation method
S68	Agarwal, Edwards and Perez-Quinones (2006)	2006	SIGCSE	experimental
S69	Desai, Janzen and Clements (2009)	2009	SIGCSE	experimental
S70	Barbosa *et al.* (2008)	2008	FIE	not applicable

B.4 Programming assignments

The topic *programming assignments* includes the papers listed in Table 41, which discuss guidelines to conduct programming assignments that include testing practices.

Table 41 – Selected papers mapped to topic “programming assignments”

study ID	reference	year	venue name	evaluation method
S71	Marrero and Settle (2005)	2005	SIGCSE	experience report
S72	Pieterse and Liebenberg (2017)	2017	Koli Calling	experimental

(continued in the next page)

Selected papers mapped to topic “programming assignments”(continued)

study ID	reference	year	venue name	evaluation method
S73	Teusner, Hille and Hagedorn (2017)	2017	FIE	experimental
S74	Jones (2000a)	2000	ADMI	not applicable
S75	Ghafarian (2001)	2001	Journal of Computing Sciences in Colleges	not applicable
S76	Isong (2001)	2001	Journal of Computing in Small Colleges	not applicable
S77	Edwards *et al.* (2008)	2008	SIGCSE Bulletin	not applicable
S78	Kussmaul (2008)	2008	OOPSLA	not applicable
S79	Middleton (2013)	2013	Journal of Computing in Small Colleges	not applicable
S80	Middleton (2015)	2015	Journal of Computing in Small Colleges	not applicable
S81	Carbone *et al.* (2000)	2000	Australasian Conference on Computing Education	qualitative
S82	Lakanen, Lappalainen and Isomöttönen (2015)	2015	Koli Calling	qualitative
S83	Bryce (2011)	2011	Journal of Computing in Small Colleges	survey
S84	Romli, Sulaiman and Zamli (2011)	2011	Communications in Computer and Information Science	survey

B.5 Programming process

The topic *programming process* includes the papers listed in Table 42, which discuss programming processes for novices, binding the activities of programming and testing.

Table 42 – Selected papers mapped to topic “programming process”

study ID	reference	year	venue name	evaluation method
S85	Smith and Stoecklin (2001)	2001	Journal of Computing in Small Colleges	descriptive
S86	Allen, Cartwright and Reis (2003)	2003	SIGCSE	experience report
S87	Rahman and Juell (2006)	2006	CSEE&T	experience report
S88	Rahman (2007)	2007	FIE	experience report
S89	Caspersen and Kolling (2009)	2009	ACM TOCE	experience report
S90	Nino (2009)	2009	Journal of Computing in Small Colleges	experience report
S91	Paul (2016)	2016	Journal of Computing in Small Colleges	experience report
S92	Edwards (2003d)	2003	EISTA	experimental
S93	Erdogmus, Morisio and Torchiano (2005)	2005	IEEE Transactions on Software Engineering	experimental
S94	Janzen and Saiedian (2006a)	2006	CSEE&T	experimental
S95	Mendonca, Guerrero and Costa (2009)	2009	FIE	experimental
S96	Buffardi and Edwards (2012b)	2012	International Journal of Information and Computer Science	experimental
S97	Neto *et al.* (2013)	2013	ICSE	experimental
S98	Camara and Silva (2016)	2016	SIGCSE	experimental
S99	Parodi *et al.* (2016)	2016	CLEI	experimental
S100	Missiroli, Russo and Ciancarini (2017)	2017	COMPSAC	experimental
S101	Scatalon *et al.* (2017b)	2017	CSEE&T	experimental
S102	Jones (2004)	2004	Journal of Computing in Small Colleges	literature review
S103	Desai, Janzen and Savage (2008)	2008	SIGCSE Bulletin	literature review

(continued in the next page)

Selected papers mapped to topic “programming process”(continued)

study ID	reference	year	venue name	evaluation method
S104	Parrish *et al.* (2000)	2000	Southeast Regional Conference	not applicable
S105	Caspersen and Kolling (2006)	2006	OOPSLA	not applicable
S106	Hundley (2010)	2010	Southeast Regional Conference	not applicable
S107	Bennedsen and Caspersen (2005)	2005	SIGCSE	qualitative
S108	Keefe, Sheard and Dick (2006)	2006	Australasian Conference on Computing Education	qualitative
S109	Murphy *et al.* (2008)	2008	SIGCSE	qualitative
S110	VanDeGrift *et al.* (2011)	2011	SIGCSE	qualitative
S111	Pearce, Nakazawa and Heggen (2015)	2015	Journal of Computing in Small Colleges	qualitative

B.6 Tools

The topic *tools* includes the papers listed in Table 43, which investigate supporting tools for the integration of testing into programming courses.

Table 43 – Selected papers mapped to topic “tools”

study ID	reference	year	venue name	evaluation method
S112	Zeller (2000)	2000	ITiCSE	experience report
S113	Cheang *et al.* (2003)	2003	Computers&Education	experience report
S114	Leal and Silva (2003)	2003	Software: Practice and Experience	experience report
S115	Roberts and Verbyla (2003)	2003	Australasian Conference on Computing Education	experience report
S116	Venables and Haywood (2003)	2003	Australasian Conference on Computing Education	experience report
S117	Choy *et al.* (2005)	2005	Advances in Web-Based Learning	experience report
S118	Collofello and Vehathiri (2005)	2005	FIE	experience report
S119	Higgins *et al.* (2005)	2005	Journal of Computing in Small Colleges	experience report
S120	Baldwin, Crupi and Estrellado (2006)	2006	SIGCSE Bulletin	experience report
S121	Fischer and Gudenberg (2006)	2006	PPPJ	experience report
S122	etteberg and Aalberg (2006)	2006	OOPSLA	experience report
S123	Elbaum *et al.* (2007)	2007	ICSE	experience report
S124	Gotel, Scharff and Wildenberg (2007)	2007	PPPJ	experience report
S125	Amelung, Forbrig and Rösner (2008)	2008	ITiCSE	experience report
S126	Tremblay *et al.* (2008)	2008	Software: Practice and Experience	experience report
S127	Proulx and Jossey (2009a)	2009	PPPJ	experience report
S128	Sant (2009)	2009	SIGCSE	experience report
S129	Tiantian *et al.* (2009)	2009	ETCS	experience report
S130	Lappalainen *et al.* (2010)	2010	ITiCSE	experience report
S131	Striewe and Goedicke (2011)	2011	ITiCSE	experience report
S132	Tremblay and Lessard (2011)	2011	ITiCSE	experience report
S133	Petit, Gimenez and Roura (2012)	2012	SIGCSE	experience report
S134	Rajaguru *et al.* (2012)	2012	ICAESM	experience report
S135	Zanden *et al.* (2012)	2012	Journal of Computing in Small Colleges	experience report

(continued in the next page)

Selected papers mapped to topic “tools”(continued)

study ID	**reference**	**year**	**venue name**	**evaluation method**
S136	Pieterse (2013)	2013	CSERC	experience report
S137	Sioson (2013)	2013	IISA	experience report
S138	Vihavainen *et al.* (2013)	2013	ITiCSE	experience report
S139	Edwards (2014)	2014	Learning Scale Conference	experience report
S140	Marcos-Abed (2014b)	2014	Western Canadian Conference on Computing Education	experience report
S141	Tillmann *et al.* (2014)	2014	ISSTA	experience report
S142	Bishop *et al.* (2015)	2015	ICSE	experience report
S143	Bradshaw (2015)	2015	SIGCSE	experience report
S144	Ishihara and Funabiki (2015)	2015	IIAI	experience report
S145	Ureel and Wallace (2015)	2015	FIE	experience report
S146	Gao, Pang and Lumetta (2016)	2016	ITiCSE	experience report
S147	Herout and Brada (2016)	2016	CSEE&T	experience report
S148	Kyrilov and Noelle (2016)	2016	Journal of Computing in Small Colleges	experience report
S149	Spacco *et al.* (2005)	2005	International Workshop on Mining Software Repositories	experimental
S150	Odekirk-Hash and Zachary (2001)	2001	SIGCSE	experimental
S151	Daly and Horgan (2004)	2003	IEEE Transactions on Education	experimental
S152	Sauvé, Neto and Cirne (2006)	2006	International Workshop on Automation of Software Test	experimental
S153	Thornton *et al.* (2008)	2008	SIGCSE	experimental
S154	Allevato, Edwards and Perez-Quinones (2009)	2009	SIGCSE	experimental
S155	Cardell-Oliver *et al.* (2010)	2010	Australian Software Engineering Conference	experimental
S156	Clarke *et al.* (2010)	2010	OOPSLA	experimental
S157	Denny *et al.* (2011)	2011	SIGCSE	experimental
S158	Dvornik *et al.* (2011)	2011	CSEE&T	experimental
S159	Enstrom *et al.* (2011)	2011	FIE	experimental
S160	Nishimura, Kawasaki and Tominaga (2011)	2011	ITHET	experimental
S161	Souza, Maldonado and Barbosa (2011)	2011	CSEE&T	experimental
S162	Wang *et al.* (2011)	2011	Computers & Education	experimental
S163	Allevato and Edwards (2012)	2012	SIGCSE	experimental
S164	Edwards *et al.* (2012)	2012	SIGCSE	experimental
S165	Kaushal and Singh (2012)	2012	AICERA	experimental
S166	Buffardi and Edwards (2013b)	2013	SIGCSE	experimental
S167	Janzen, Clements and Hilton (2013)	2013	ICSE	experimental
S168	Jezek, Malohlava and Pop (2013)	2013	CSEE&T	experimental
S169	Shams (2013a)	2013	SPLASH	experimental
S170	Shams and Edwards (2013)	2013	ICER	experimental
S171	Vujosevic-Janicic *et al.* (2013)	2013	Information and Software Technology	experimental
S172	Allevato and Edwards (2014)	2014	Software - Practice and Experience	experimental
S173	Buffardi and Edwards (2014b)	2014	ITiCSE	experimental
S174	Edwards, Shams and Estep (2014)	2014	SIGCSE	experimental
S175	Politz *et al.* (2014)	2014	ITiCSE	experimental
S176	Rubio-Sanchez *et al.* (2014)	2014	Computers in Human Behavior	experimental
S177	Souza *et al.* (2014)	2014	FIE	experimental
S178	Blaheta (2015)	2015	SIGCSE	experimental
S179	Buffardi and Edwards (2015)	2015	SIGCSE	experimental

(continued in the next page)

Selected papers mapped to topic “tools”(continued)

study ID	**reference**	**year**	**venue name**	**evaluation method**
S180	Reynolds *et al.* (2015)	2015	Journal of Computing in Small Colleges	experimental
S181	Souza, Isotani and Barbosa (2015)	2015	International Journal of Knowledge and Learning	experimental
S182	Earle, Fredlund and Hughes (2016)	2016	ITiCSE	experimental
S183	Birch, Fischer and Poppleton (2016)	2016	ITiCSE	experimental
S184	Braught and Midkiff (2016)	2016	SIGCSE	experimental
S185	Rajala *et al.* (2016)	2016	Australasian Computer Science Week Multiconference	experimental
S186	Sridhara *et al.* (2016)	2016	Learning Scale Conference	experimental
S187	Tang *et al.* (2016)	2016	ITiCSE	experimental
S188	Smith *et al.* (2017)	2017	ITiCSE	experimental
S189	Madeja and Poruban (2017)	2017	International Scientific Conference on Informatics	experimental
S190	Yi *et al.* (2017)	2017	Joint Meeting on Foundations of Software Engineering	experimental
S191	Krusche and Seitz (2018)	2018	SIGCSE	experimental
S192	Ala-Mutka (2005)	2005	Computer Science Education	literature review
S193	Douce, Livingstone and Orwell (2005)	2005	Journal of Computing in Small Colleges	literature review
S194	Ihantola *et al.* (2010)	2010	Koli Calling	literature review
S195	Romli, Sulaiman and Zamli (2010)	2010	International Symposium on Information Technology	literature review
S196	Allevato *et al.* (2008)	2008	Educational Data Mining	not applicable
S197	Gustafson and Dwyer (2000)	2000	FIE	not applicable
S198	Jackson (2000)	2000	ITiCSE	not applicable
S199	Allen, Cartwright and Stoler (2002)	2002	SIGCSE	not applicable
S200	Higgins, Symeonidis and Tsintsifas (2002)	2002	ITiCSE	not applicable
S201	Andrianoff *et al.* (2003)	2003	OOPSLA	not applicable
S202	Jones and Allen (2003)	2003	ITiCSE	not applicable
S203	Morris (2003)	2003	FIE	not applicable
S204	Patterson, Kölling and Rosenberg (2003)	2003	ITiCSE	not applicable
S205	Proulx and Rasala (2004)	2004	SIGCSE Bulletin	not applicable
S206	Spacco, Hovemeyer and Pugh (2004)	2004	OOPSLA	not applicable
S207	Sun and Jones (2004)	2004	Southeast Regional Conference	not applicable
S208	Allowatt and Edwards (2005)	2005	OOPSLA	not applicable
S209	Feng and McAllister (2006)	2006	FIE	not applicable
S210	Spacco *et al.* (2006)	2006	OOPSLA	not applicable
S211	Helmick (2007)	2007	SIGCSE	not applicable
S212	Ihantola (2007)	2007	Informatics in Education	not applicable
S213	Murphy and Yildirim (2007)	2007	FIE	not applicable
S214	Tremblay, Laforest and Salah (2007)	2007	SIGCSE	not applicable
S215	Edwards and Perez-Quinones (2008)	2008	ITiCSE	not applicable
S216	Fu *et al.* (2008)	2008	SIGCSE	not applicable
S217	Rossling and Hartte (2008)	2008	ITiCSE	not applicable
S218	Proulx and Jossey (2009b)	2009	SIGCSE	not applicable
S219	Clements and Janzen (2010)	2010	ICST	not applicable

(continued in the next page)

Selected papers mapped to topic “tools”(continued)

study ID	**reference**	**year**	**venue name**	**evaluation method**
S220	Karavirta and Ihantola (2010)	2010	ITiCSE	not applicable
S221	Leal and Silva (2010)	2010	book	not applicable
S222	Ricken and Cartwright (2010)	2010	SIGCSE	not applicable
S223	Bell, Sheth and Kaiser (2011)	2011	Int. Workshop on Social Software Engineering	not applicable
S224	Hull, Powell and Klein (2011)	2011	ITiCSE	not applicable
S225	Sheth, Bell and Kaiser (2011)	2011	Int. Workshop on Games and Software Engineering	not applicable
S226	Snyder, Edwards and Perez-Quinones (2011)	2011	SIGCSE	not applicable
S227	Zimmerman, Kiniry and Fairmichael (2011)	2011	CSEE&T	not applicable
S228	Llana, Martin-Martin and Pareja-Flores (2012)	2012	Koli Calling	not applicable
S229	Danutama and Liem (2013)	2013	Procedia Technology	not applicable
S230	Tillmann *et al.* (2013)	2013	ASE	not applicable
S231	Zhu and Chen (2013)	2013	Journal of Software	not applicable
S232	Zingaro *et al.* (2013)	2013	SIGCSE	not applicable
S233	Akour (2014)	2014	CSCI	not applicable
S234	Marcos-Abed (2014a)	2014	ITiCSE	not applicable
S235	O’Brien, Goldman and Miller (2014)	2014	Learning Scale Conference	not applicable
S236	Pribela *et al.* (2014)	2014	CEUR	not applicable
S237	Brian *et al.* (2015)	2015	ITiCSE	not applicable
S238	Combefis and Paques (2015)	2015	Workshop on Educational Software Engineering	not applicable
S239	Pietrikova, Juhar and Stastna (2015)	2015	ICETA	not applicable
S240	Pozenel, Furst and Mahnic (2015)	2015	MIPRO	not applicable
S241	Turner (2015)	2015	SIGCSE	not applicable
S242	Xie *et al.* (2015)	2015	Int. Workshop on CrowdSourcing in Software Engineering	not applicable
S243	Johnson (2016)	2016	SIGCSE	not applicable
S244	Lippe *et al.* (2016)	2016	SIGPLAN Symposium on Scala	not applicable
S245	Wilcox (2016)	2016	SIGCSE	not applicable
S246	Funabiki *et al.* (2017)	2017	AINA	not applicable
S247	Clegg, Rojas and Fraser (2017)	2017	ICSE	not applicable
S248	Dewey *et al.* (2017)	2017	ITiCSE	not applicable
S249	Joy, Griffiths and Boyatt (2005)	2005	Journal of Computing in Small Colleges	qualitative
S250	Saikkonen, Malmi and Korhonen (2001)	2001	ITiCSE	survey
S251	Harris, Adams and Harris (2004)	2004	Journal of Computing in Small Colleges	survey
S252	Juedes (2005)	2005	FIE	survey
S253	Shaffer (2005)	2005	SIGCSE Bulletin	survey
S254	Spacco *et al.* (2006)	2006	SIGCSE	survey
S255	Nordquist (2007)	2007	Journal of Computing in Small Colleges	survey
S256	Suleman (2008)	2008	SAICSIT	survey
S257	Brown *et al.* (2012)	2012	ITiCSE	survey
S258	Johnson (2012)	2012	ITiCSE	survey
S259	Funabiki, Nakamura and Kao (2014)	2014	GCCE	survey

(continued in the next page)

Selected papers mapped to topic “tools”(continued)

study ID	reference	year	venue name	evaluation method
S260	Pape *et al.* (2016)	2016	ICSE	survey
S261	Valle *et al.* (2017)	2017	FIE	survey

B.7 Program/test quality

The topic *program/test quality* includes the papers listed in Table 44, which investigate assessment of students’ submitted code (program or tests).

Table 44 – Selected papers mapped to topic “program/test quality”

study ID	reference	year	venue name	evaluation method
S262	Steinberg (2001)	2001	XP Universe Conference	descriptive
S263	Morisio, Torchiano and Argentieri (2004)	2004	International Symposium on Software Metrics	experimental
S264	Aaltonen, Ihantola and Seppala (2010)	2010	OOPSLA	experimental
S265	Brito *et al.* (2012)	2012	CLEI Electronic Journal	experimental
S266	Utting *et al.* (2013b)	2013	ITiCSE	experimental
S267	Edwards and Shams (2014a)	2014	ICSE	experimental
S268	Whalley and Kasto (2014)	2014	Australasian Computing Education Conference	experimental
S269	Lemos *et al.* (2015)	2015	Int. Symposium on Software Reliability Engineering	experimental
S270	Shams and Edwards (2015)	2015	SIGCSE	experimental
S271	Edwards and Li (2016)	2016	Koli Calling	experimental
S272	Gómez, Vegas and Juristo (2016)	2016	ICSE	experimental
S273	Lemos *et al.* (2017)	2017	Journal of Systems and Software	experimental
S274	Souza, Kolling and Barbosa (2017)	2017	FIE	experimental
S275	Shams (2013b)	2013	ICER	not applicable
S276	Luxton-Reilly *et al.* (2013)	2013	ITiCSE	qualitative

B.8 Concept understanding

The topic *concept understanding* includes the papers listed in Table **??**, which investigate the assessment of students’ knowledge of programming and testing concepts.

Table 45 – Selected papers mapped to topic “concept understanding”

study ID	reference	year	venue name	evaluation method
S277	Sanders *et al.* (2013)	2013	ITiCSE	experimental
S278	Luxton-Reilly *et al.* (2017)	2017	ITiCSE	exploratory

B.9 Perceptions/behaviors

The topic *perceptions/behaviors* includes the papers listed in Table **??**, which investigate students’ attitudes towards software testing.

Table 46 – Selected papers mapped to topic “perceptions and behaviors”

study ID	**reference**	**year**	**venue name**	**evaluation method**
S279	Spacco and Pugh (2006)	2006	OOPSLA	experimental
S280	Janzen and Saiedian (2007)	2007	ICSE	experimental
S281	Buffardi and Edwards (2012a)	2012	ITiCSE	experimental
S282	Buffardi and Edwards (2013a)	2013	ICER	experimental
S283	Fidge, Hogan and Lister (2013)	2013	Australasian Computing Education Conference	experimental
S284	Spacco *et al.* (2013)	2013	ITiCSE	experimental
S285	Buffardi and Edwards (2014a)	2014	SIGCSE	experimental
S286	Edwards and Shams (2014b)	2014	ITiCSE	experimental
S287	Baumstark Jr. and Orsega (2016)	2016	Journal of Computing in Small Colleges	experimental
S288	Kolikant (2005)	2005	ICER	qualitative
S289	Thompson, Hunt and Kinshuk (2006)	2006	Australasian Conference on Computing Education	qualitative
S290	Kolikant and Mussai (2008)	2008	Computer Science Education	qualitative
S291	Mendonca *et al.* (2009)	2009	FIE	qualitative
S292	Pham *et al.* (2014)	2014	Symposium on the Foundations of Software Engineering	qualitative
S293	Ize, Pope and Weerasinghe (2017)	2017	ITiCSE	qualitative

ICMC USP
SÃO CARLOS