Journal of Computer Engineering
ISSN 10003428

Original Research Article
Vol 12, Issue 12, 2023 | ISSN: 1000-3428

Exploring the Efficacy of Software Engineering-Centric Instructional Models in End-User Software Development

Alessia Bianchi, Elena Marquez, and Riccardo Zucchelli

University of Pisa, Italy (Alessia Bianchi), University of Seville, Spain (Elena Marquez), University of Verona, Italy (Riccardo Zucchelli)

Abstract

Aim/Purpose	This work aims to introduce and evaluate an instructional strategy that aids end-users with developing their software products during intensive projectbased events.
Background	End-users produce software in the labor market, and one of the challenges for End-User Software Engineering (EUSE) is the need to create functional software products without a formal education in software development.
Methodology	In this work, we present an instructional strategy to expose end-users to Agile-based Software Engineering (SE) practices and enhance their ability to developing high-quality software. Moreover, we introduce a SE approach for the collection of metrics to assess the effectiveness of the instructional strategy. We conducted two case studies to validate the effectiveness of our strategy; the comprehensive analysis of the outcome products evaluates the strategy and demonstrates how to interpret the collected metrics.
Contribution	This work contributes to the research and practitioner body of knowledge by leveraging SE centric concepts to design an instructional strategy to lay the foundations of SE competencies in inexperienced developers. This work presents an instructional strategy to develop SE competencies through an intensive and time-bound structure that may be replicated. Moreover, the present

Indexed keywords: Computer Engineering, Advanced Computing, Technology, Open Access

Article History: Received: 02 October 2023 | Accepted: 25 November 2023 | Published: 07 December 2023

work introduces a framework to evaluate these competencies from a product-centric approach, specialized for non-professional individuals. Finally, the framework contributes to understanding how to assess software quality when the software product is written in non-conventional, introductory programming languages.

© 2023 The Authors. Open Access under CC BY 4.0.
How to cite: Alessia Bianchi, Elena Marquez, and Riccardo Zucchelli (2023). Exploring the Efficacy of Software Engineering-Centric Instructional Models in End-User Software Development. Journal of Computer Engineering, 12(12), 58–80. DOI:
https://doi.org/10.5281/zenodo.19351073

Findings	The results show the effectiveness of our instructional strategy: teams were successful in constructing a working software product. However, participants did not display a good command of source code order and structure.
Recommendations for Practitioners	Our instructional strategy provides practitioners with a framework to lay for foundations in SE competencies during intensive project-based events. Based on the results of our case studies, we provide a set of recommendations for educational practice.
Recommendations for Researchers	We propose an assessment framework to analyze the effectiveness of the instructional strategy from a SE perspective. This analysis provides an overall picture of the participants' performance; other researchers could use our framework to evaluate the effectiveness of their activities, which would contribute to increasing the possibility of comparing the effectiveness of different instructional strategies.
Impact on Society	Given the number of end-user developers who create software products without a formal SE training, several professional and educational contexts can benefit from our proposed instructional strategy and assessment framework.
Future Research	Further research can focus on improving the assessment framework by including both process and product metrics to shed light on the effectiveness of the instructional strategies.

Keywords	intensive project-based events, bootcamp, end-user software engineering, EUSE, instructional strategy, assessment framework

INTRODUCTION

Computer programming is a widespread practice; in consequence, not only specialists in Software Engineering (SE) or Computer Science (CS) but also end-users produce software for the labor market. The term *end-user* was initially introduced to distinguish those who purely use software systems from professional developers. Now, it refers to people (such as secretaries, accountants, teachers) who develop software as part of their professional practice, without having either a degree in CS/SE or extensive experience in software development (Burnett & Myers, 2014; Costabile et al., 2008; Ko et al., 2011; Ye & Fischer, 2007). In 2005, there were about three million professional programmers in the U.S. (Scaffidi et al., 2005), and over 12 million more people said they were programming at work even though it was not their job description. In 2015, 7 (out of 26) million U.S. online job postings valued coding as a technical skill (Burning Glass Technologies, 2016). In 2018, a survey found that end-users play a critical role in creating customer-facing (24%) and enterprise apps (22%) (Warren, 2018).

The downside is the overall low quality of end-user-created software, in part due to the lack of specific training in SE (Scheubrein, 2003). Even if the errors are non-catastrophic, their effects can have an impact on a production environment. For example, web applications created by small-business owners to promote their businesses can result in loss of revenue and credibility if they contain pages that are displayed incorrectly (Burnett, 2009).

In 2000, Shaw claimed that software development should be treated from an engineering perspective for all students who learn software development. Afterwards, SE and Education have been the focus of extensive research, especially at the undergraduate level

(Kastl et al., 2016). Moreover, End-User Software Engineering (EUSE) has been established as an area of interest within SE to increase the quality of end-user-created software, by focusing on the entire software lifecycle (Burnett, 2009). The challenge of EUSE is incorporating SE activities into end-users' existing workflow by taking into account the factors that determine diversity, including learning styles, profiles, interests, situations (Chimalakonda & Nori, 2013), and cultural factors (Frieze et al., 2006).

People outside SE see the benefits of coding for the creation of their app startups, and many are looking for intensive project-based experiences (e.g., bootcamps, hackathons, summer schools), which are getting more and more popular (Decker et al., 2015). A survey on the rise of coding bootcamps found a 175% growth rate for the programming bootcamp market in 2014 alone (Champagne, 2016). EUSE challenges are also present during these events. For example, participants focus on their domain-specific goals and not on learning SE (Chimalakonda & Nori, 2013). Moreover, since EUSE addresses software quality issues, a challenge is posed to understand how to assess software quality, in particular, when using introductory (e.g., block-based) programming languages.

In this paper, we first describe an *instructional strategy* to lay foundations in SE competencies during bootcamps. The goal is to foster eXtreme Programming (XP) practices, which are the right candidate for end-users (Fronza et al., 2019), without changing the existing workflow or the participants' priorities (Fronza & Pahl, 2018a). Moreover, we propose an *assessment framework* to analyze the outcomes from a SE perspective; the framework provides an overall picture of the participants' performance and helps instructors/researchers learn from individual cases. Finally, we describe the results of two case studies that demonstrate the effectiveness of the proposed instructional strategy and show how to interpret the collected metrics.

The remaining of the paper is organized as follows: Section *Background and Related Work* reviews background literature and current state of the art in the subject matter; Section *Research Methodology* outlines our strategy to provide answers to the posed questions. Section *Instructional Strategy* details our strategies to promote SE practices during bootcamps; Section *Assessment Framework* describes the strategy to analyze the outcome of the bootcamps from a SE perspective. Section *Case Studies* describes two observed populations, and Section *Results* reports the outcome of the two case studies. Section *Discussion* provides a complete elaboration of the findings and limitations of this study. Finally, Section *Conclusions* closes this work, provides recommendations for educational practice and directions for further research.

BACKGROUND AND RELATED WORK

Software Engineering and Education have been the focus of extensive research, especially on the value that can be acquired in a classroom setting toward enabling practitioners to develop successful products. Shaw (2000) established challenges and aspirations for educators in SE, including fostering current practices in an ever-changing arena and

having a clear focus on practical skills. Since then, classroom experiences efficient in mapping software processes to course sessions and deliverables have been published, as well as research that discusses practices, behaviors, and interactions among students (Liebenberg et al., 2015).

Curriculum design on SE is a frequent object of study, analyzing the timeliness and relevance of courses and practical education, particularly at the undergraduate level. This body of research focuses on developing professional competence on the necessary skills to abstract real-world problems and deliver solutions in the form of software products. The professional segments to which these efforts are directed are software engineers, computer scientists, and information technology experts.

From a practitioner's perspective, it is not uncommon that people need to develop software during their work activities without having a CS/SE degree or even experience in software development (Fronza & Pahl, 2019; Paternò, 2013; Scaffidi et al., 2005). They are called *end-users* and have different intent respect to professional software developers (Burning Glass Technologies, 2016): they write programs to support their work (Ko et al., 2011) and do not consider quality as a primary concern.

This merge between roles is not new. For instance, in the late 2000s, the term *DevOps* was coined to represent a SE practice, where software development and software operation work together to deliver software continuously. Thanks to this combination, the business can seize emerging and existing market opportunities and reduce the time required for including the client's feedback (Kamuto & Langerman, 2017). The term DevOps defines software development professionals who expanded their role to include the actual software operation. They take the role of the end-user and put the proficiency of software development to the service of better understanding the software product in its real context of use. Therefore, DevOps extends the software development skills to a level of understanding of the domain, which involves knowledge of business, market, science, and others.

End-users take the opposite approach. Professionals in a specific domain of business (market, science, technique, and others) expand their knowledge to gain skills in software development to personally develop software tools (of different levels of complexity) for the daily execution of their jobs, regardless of the discipline or context.

End-user software development takes advantage of a series of standard tools. For instance, Microsoft Office offers the capacity of automating repetitive tasks using macros that can be recorded directly from the user interface, i.e., without coding. The users who need more complex features can write simple code using the Visual Basic language. Many other products enable users to start developing simple software pieces without formally learning software development. For example, a survey of 129,130 App Inventor users found that 73% of respondents used App Inventor at home, not in a formal learning environment (Xie et al., 2015).

In the early 2010s, initiatives to attract interest and talent to software development became very popular. The approach of hackathons, bootcamps, and crash courses is *hands-on*, that is, focused on practical aspects of programming languages and working software. Facilitators rarely invest time in explaining fundamental principles of software design, implementation, and validation. Concepts like *example centric programming* or *copy-paste programming* have become popular, and researchers investigated their

impact on the practice and quality of the final products (Hou et al., 2009). More recently, researchers have tried to define how hackathons should be organized and structured to obtain the best out of these experiences (Gama et al., 2018; Lara & Lockwood, 2016). Moreover, they investigated more deeply the educational advantages of hackathons and similar experiences when teaching SE concepts (Gama, 2019; Uys, 2019).

EUSE also spans domain-specific applications. For instance, control systems engineers create closedloop systems to keep control of several parameters in physical systems. As a consequence, control engineers become software engineers when they implement control algorithms in the form of embedded software products thanks to development tools like LabView or Simulink, in which signal or data flow diagrams eventually turn into C code.

Moreover, with the rise of open-community programming camps, such as hackathons or bootcamps, the opportunity to implement software to solve a given challenge is brought to an open community (Decker et al., 2015; Porras et al., 2005, 2018). Hackathons are events in which teams spend a limited number of hours to accomplish a given goal through software development; however, their scope spans in several other fields, such as electronic design, robotics, or maker culture.

The initiative of empowering any user to develop working software is praiseworthy and may contribute to the accomplishment of countless business objectives; nonetheless, it poses a relevant challenge to the research and practice of SE and software quality assurance. The approach of EUSE involves systematic and disciplined activities that address software quality issues, but these activities are secondary to the goal that the program is helping to achieve. EUSE aims at finding ways to incorporate SE activities into end-users' existing workflow, respecting the nature of their work and their priorities. Diversified contexts, backgrounds, needs, and cultural factors (Frieze et al., 2006) take a toll on the instructional strategy and the quality characteristics of the outcome products (Chimalakonda & Nori, 2013). A challenge is understanding how to achieve these goals during intensive project-based events (e.g., bootcamps), which usually attract audiences with different backgrounds and needs.

Hence, the following research questions are raised:

- **RQ1:** How to adapt a Software Engineering instructional strategy to a diversified set of audiences with different backgrounds and needs during intensive project-based events?
- **RQ2:** How should we assess Software Quality, especially when considering non-conventional (e.g., block-based) development tools? What metrics apply in this case?

The democratization of software development via integrated development tools, graphic user interfaces, and block-based programming languages, as well as the prominence of intensive and practical training experiences, open a promising arena of end-user-created products in the short- and mid-run. Moreover, as far as the underlying research of the

present paper can be, there is a lack of scholarly literature that assists in the process of developing and appraising high-quality end-user-created products. With this motivation, our research questions aim to shed light on an emerging yet highly relevant research subject matter. In the next section, we outline a research methodology to provide answers to the posed questions.

RESEARCH METHODOLOGY

Since we aim to answer questions that relate directly to the learning and developmental practice, we opt to go with an applied research methodology, in which we take principles of SE and put them in practice towards the design of an *instructional strategy* that builds on top of solid SE practices. To do this, we will review instructional strategies, documented practices, and implementation environments.

Our research is quantitative as we aim to propose an *assessment framework* for evaluating software quality. Assessing the software product in its different characteristics implies the collection and interpretation of metrics that support the understanding of a product and deliver insightful information about its nature. Having quantitative information permits as well to relate characteristics of the product with the SE instructional strategy outlined first. To accomplish it, we investigate and implement well-grounded metric collections, data interpretation, and visualizations that can deliver the necessary information to understand the software product. The conjunction of an applied and quantitative research practice will result in an explanatory study that, on top of a case study involving different populations participating in a project-based software development event, provides the necessary elements to understand better the proposed strategy, the assessment framework, and the software products delivered after working in different contexts. Our goal is to observe two independent heterogeneous groups of participants with relatively limited knowledge in software development, which represent a typical intensive project-based event, i.e., participants of our case studies come from a variety of ages, cultures (Frieze et al., 2006; Sammut-Bonnici & McGee, 2015), and backgrounds. Therefore, as we will detail in the *Instructional Strategy* section, we observe separate groups, with the independence of placement, age, and background.

INSTRUCTIONAL STRATEGY

One of the challenges in the EUSE field is understanding how to incorporate SE activities into intensive project-based events (e.g., bootcamps), which usually attract audiences with different backgrounds and needs. Hence, our instructional strategy is based on an intensive project-based event that simulates a professional environment in which participants develop applications for mobile devices (e.g., cellular phones or tablets) operated by the Android OS (Fronza et al., 2016). The event consists of 20 hours of activity, divided into five sessions (Table 1). We provide each team/person with a table, one computer per person, at least one mobile device (some participants bring their own devices), materials for project management, such as whiteboards, pens, paper, and post-its.

Table 1. Timetable of the proposed intensive project-based event.

SESSION	HOURS	ACTIVITIES
		Foundations of logical thinking, structured sequencing, and data abstraction; Preparatory activities (e.g., setting of the programming environment).
2 - 4		Development of a mobile app by iterations: problem definition; design of a solution; development iterations.
		Completing and polishing the product towards a final presentation

Our goal (RW1) is understanding how SE instructional strategies can be adapted to different audiences during intensive project-based events. In particular, we consider two types of populations (Table 2), namely High School students (HS) and University Postgraduate students (UP).

Table 2. Characteristics of the two considered audiences.

PARTICIPANTS	CHARACTERISTICS
High school students (HS)	• 15-18 years old • little or no previous knowledge of software development • attending different schools, from non-vocational to computer science • they want to choose their future career
University postgraduate students (UP)	• 25-50 years old • no previous knowledge of software development • they need to develop code as part of their career (e.g., to create web pages) • they have a specific idea that they want to develop during the activity

Agile methods accommodate end-users' working style (Fronza, et al., 2017; Kastl et al., 2016; Meerbaum-Salant & Hazzan, 2010), which is preferably collaborative (Costabile et al., 2008), opportunistic, incremental, and by trial-and-error phases (Burnett & Myers, 2014). Thus, we select the following XP practices by considering participants' characteristics and activities focus (i.e., programming, process, team) (Fronza et al., 2019):

- HS: In the K-12 context, it is advisable to emphasize the process by which students arrive at the product (Steghöfer et al., 2016). Thus, we focus more on practices related to process and team, i.e., user stories, small releases, system metaphor and coding standard, collective ownership, testing, pair programming, continuous integration, and on-site customer.
- UP: Each participant has a specific idea that she/he wants to develop (e.g., to address a particular customer's need). Thus, we prefer individual work, and we focus on process and programming practices, i.e., user stories, small releases,

metaphor, and coding standard, simple design, refactoring, testing, and on-site customer.

Under consideration of the underlying EUSE principles, we neither try to impose practices nor add specific lessons. Instead, we adopt a set of strategies/activities to let participants reflect and reason action courses when there is a need for planning, managing, or empowering. Participants heuristically mix these activities as needed, depending on the SE phase they are working on (e.g., design, testing). Our *learn-by-playing approach* differs from gamification (Becker, 2015) in the aspect that participants do not follow a game-like journey to accomplish the goal of the bootcamp. Games are proposed to the group in selected segments of the bootcamp, to exemplify or drive a specific behavior that will be beneficial for the rest of the activities; for this reason, we dedicate 5-10 minutes to reflecting on the takeaway messages. Table 3 maps each strategy to the corresponding XP practices. Some strategies are used only with a specific audience. For instance, *System Metaphor* and *Simple Design* are used with the UP participants to leverage their previous knowledge about several stages in the creative process, which are relevant for the software development processes.

Manipulatable examples. Participants are generally open to expanding their software development skills, but they do not expect a programming course. For this reason, we help them explore their design ideas by using manipulatable examples (Burnett & Myers, 2014), from the perspective of learning-by-doing. Moreover, we let them create new configurations and designs by tailoring software components in their software environments.

Focus on the problem-solving activity. We support an opportunistic and incremental working style (Burnett & Myers, 2014), and we focus on the problem-solving activity rather than on the SE lifecycle. This way, participants can heuristically mix reverse engineering, reuse, programming, testing, and debugging, mostly by trial-and-error (Burnett & Myers, 2014).

Alert without imposing. Participants are usually not concerned with dependability problems. Thus, we alert them to dependability problems, and we assist them with their explorations into those problems to whatever extent they choose to pursue such explorations to refactor their solutions. We present comments on product quality to the participants during the final presentation of the project, along with some suggestions for further quality improvement.

We are here to help. Participants can ask for our support whenever needed by first showing their intermediate product (current release) and describing their issue together with the solutions that they already tested. Using this strategy, we aim at fostering teams' self-organization on their projects by reducing their dependence on the instructor's assistance (Kastl et al., 2016).

Block-Based Programming Language (BBPL). MIT App Inventor (Wolber et al., 2011) is a

BBPL for mobile app development, which counts over 11 million users and 48 million created apps (MIT App Inventor, 2020). App Inventor allows problem-driven learning (Morelli et al., 2011) and can be used to foster XP practices (Fronza et al., 2019), such as a) *continuous integration*, as it forces users to integrate the new features on top of the

existing ones, and b) *refactoring* and *testing* because each functionality gets immediately tested to see if the added blocks work correctly.

Table 3. A mapping between strategies, XP practices, and participants.

STRATEGY	PRACTICE	PARTICIPANTS
Manipulatable examples	User stories	HS, UP
Focus on the problem-solving activity	Small releases, testing	HS, UP
Alert without imposing	Refactoring, Testing	HS, UP
We are here to help	Small releases, teamwork, on-site customer	HS, UP
Block-Based Programming	Continuous integration, refactoring, testing	HS, UP
Teamwork	Collective ownership, pair programming, metaphor and coding standard	HS
Marshmallow challenge	Prototyping and iterating, quick collaboration, simple design, teamwork	HS
Tell me how you make toast	Simple design, teamwork, user stories	HS
STRATEGY	PRACTICE	PARTICIPANTS
Letters with our bodies	User stories, teamwork, simple design	HS
User Persona and User Journey	System Metaphor	UP
Point of View	Simple design	UP

Teamwork. We let HS participants work in teams (of three), formed by the instructors (Oakley et al., 2004), as they enjoy communicating with friends and performing collaborative activities (Costabile et al., 2008). Each team represents an independent 'company' (Figure 1). Teams choose their name to build team coherence (Millis & Cottell, 1997); then, teams choose their projects following a challenge-based learning framework (Nichols et al., 2016). At their discretion, team members can decide to collaborate on the same code or to develop software parts individually and then integrate them. In this case, to foster collective ownership and pair programming, we frequently suggest pairing up to work on the same piece of code. Moreover, we encourage the adoption of coding standards to facilitate the integration of portions of code developed individually.

Figure 1. The organization of the room during our events.

Marshmallow challenge. During session 1, each team builds a structure in 18 minutes using 20 sticks of spaghetti, one yard of tape, one yard of string, and one marshmallow (Wujec, 2010). The winning team is the one that constructs the tallest freestanding structure with the marshmallow on top (Figure 2). The takeaway messages of this activity regard: prototyping and iterating can help achieve success, the importance of collaborating very quickly, and the value of cross-functional teams.

Tell me how you make toast. During session 2, each person sketches a diagram of how to make toast, one step per post-it (Wujec, 2013). Then, the participants combine all the individual solutions in one solution (Figure 2); to do that, they identify the common steps, discard the unnecessary ones, and so on. The takeaway message of this activity is about the importance of working together toward a solution by identifying small steps, those that ideally should be on an Agile task board.

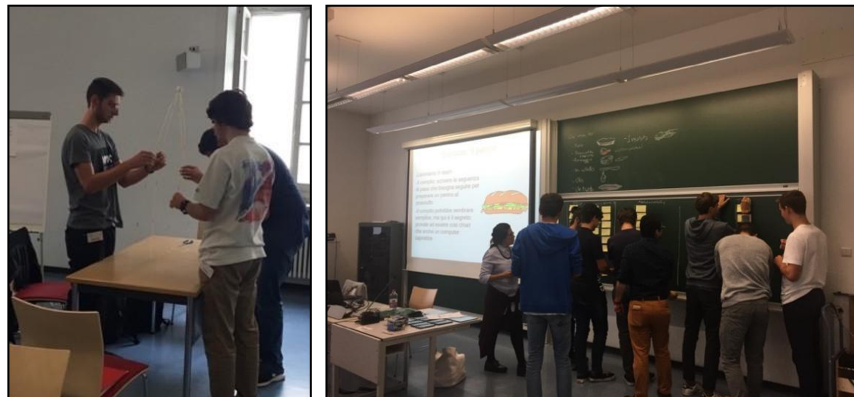

Figure 2. "Marshmallow challenge" and "tell me how you make toast" game.

Letters with our bodies. During session 3, team members use their bodies to construct letters that form the announced word. With no further instructions, they need to decide who will represent each letter, and in some instances (e.g., a letter "M"), how to set it up with more than one participant (Figure 3). Participants learn the importance of understanding ambiguous requirements, and team selforganization with little or no guidance from facilitators.

Figure 3. "Letters with our bodies" game.

User persona and user journey. An Agile system metaphor is a description of the system that can be understood by different stakeholders. To promote the adoption of this practice, we ask UP participants to create a User Persona and a User Journey. The aggregation of these two concepts has a similar purpose to the Agile system metaphor. A *User Persona* encloses an archetypical description of the user (such as actor or customer) who participates in the system's operation. A *User Journey* represents a standard language description of the process the system deals with. In this way, our instructional strategy takes advantage of processes and practices that are usual in other fields (in this case, the creative world) and put them to service a relevant stage of the software development process.

Point of view. A *Point of View* chart has the same information of a User Story used in Agile software development: *As (user), I need to (feature) In order to (goal) Because of (added value)*. Thus, participants need to identify strictly who will be interacting with the feature, what is the goal at hand, and what is the value that such feature delivers. If they experience difficulties identifying a goal or justifying an eventual added value, it means that the user story is not worthwhile to be developed, narrowing the scope of the system, motivating focus, and overall simplifying the design of the system.

ASSESSMENT FRAMEWORK

The goal of our instructional strategy is to expose participants to Agile-based SE practices to enhance their ability to developing high-quality software. For this reason, we developed a framework that includes both product and process assessment, which we detail in this section. Under consideration of the underlying principles of Project-Based Learning (PBL) (Romeike & Göttel, 2012), we did not hand out tests, and we preferred *critique and revision*, supported by observation and code inspections (Fronza et al., 2017).

PRODUCT ASSESSMENT

This part of the assessment framework analyzes, from a SE perspective, the software products developed by participants when adopting our instructional strategy. Using a BBPL prevents us from applying professional metrics and tools (Fronza & Pahl, 2018b). Indeed, in our case, source artifacts consist of blocks sorted and matched to follow a flow, execute a sentence, or evaluate a condition. Therefore, our assessment framework defines an appropriate set of metrics to analyze App Inventor projects by capitalizing on existing experiences (mainly for Scratch). Of note is the project called Dr. Scratch, which assigns a Computational Thinking (CT) score to a Scratch project and detects bad programming habits or potential errors (Moreno-Leon et al., 2015). Another project called Ninja Code Village (Ota et al., 2016) automatically assesses CT concepts in Scratch. Some research work focused on mapping the professional metrics to the BBPLs environment (Hermans & Aivaloglou, 2016). S. Grover (2017) described several difficulties novice programmers exhibit in introductory setting (e.g., assigning meaningful names to variables). Waite (2017) explored code smells in BBPLs. Hermans and Aivaloglou (2016) pursued the same goal for the specific case of Scratch. Focusing on the specific context of App Inventor, Xie et al. (2015) extracted several metrics from project data to compare App Inventor learnability and capability. Xie & Abelson (2016) adapted computational concepts for assessing CT in Scratch for use with App Inventor.

Our assessment framework capitalizes on the existing experiences. It extracts five groups of metrics to analyze App Inventor projects, from a SE perspective, namely, component metrics, computational concepts blocks, code smells, complexity metrics, and size. We use these metrics to evaluate the quality of each product and the overall software quality capacity reached by the entire group.

Component metrics. Four metrics are part of this set:

- Number of screens of the application;
- Number of components by functionality, based on the categories in the App Inventor palette (i.e., user interface, media (sound is disregarded), drawing, sensors, social, storage, connectivity) (Xie et al., 2015);
- Total Number of Components (TNC), the sum of all the components by functionality;
- The total Number of Unique Blocks (NOUB) is an indicator of the project's sophistication:

  a greater NOUB correlates with the ability to use App Inventor to create apps that have more advanced functionality (Xie et al., 2015).

Computational concepts blocks. As shown in Figure 4, we count six types of blocks to represent six computational concepts (Brennan & Resnick, 2012; Xie & Abelson, 2016).

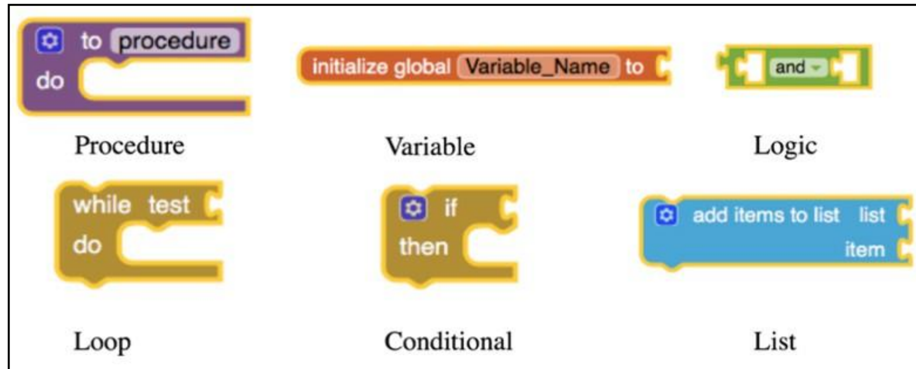

Figure 4. Computational concepts: an example of a block for each category.

Code smells. As shown in Table 4, we define code smells for App Inventor by adapting the definitions for Scratch (Grover, 2017; Hermans & Aivaloglou, 2016; Waite, 2017).

Table 4. Code smells.

METRIC	DEFINITION	REF.
Names	Percentage of components that have not been renamed	(Waite, 2017)
Superfluous stuff	Code blocks left lying around (in part/yes/no)	(Waite, 2017)
Duplication	App Inventor code suffers from the duplication smell (in part/yes/no) when similar computations or events occur in multiple places in the program (could be implemented more elegantly, for example by using a loop)	(Hermans & Aivaloglou, 2016; Waite, 2017)
Long method	App Inventor code suffers from the Long Method smell (in part/yes/no) if a group of blocks grows very large (imply a lack of decomposition and design)	(Hermans & Aivaloglou, 2016; Waite, 2017)
Variables	Variables have a meaningful name (in part/yes/no)	(Grover, 2017)

Complexity metrics. We extract two metrics as indicators of complexity: (a) Cyclomatic Complexity (CC), the number of decision points in the code (e.g., repeat until, if then) plus one (Moreno-Leon et al., 2016); and (b) the number of *when* blocks (e.g., when a button is clicked) (Fronza et al., 2020).

Size. Software size can be measured by counting the number of lines in the text of the source code. This metric is typically used as a way to judge the productivity of individual developers. However, this approach has been largely criticized. Indeed, skilled developers can develop the same functionality with far less code; in contrast, inexperienced developers often resort to code duplication, increasing the number of code lines. The two major approaches to counting lines of code are (a) physical LOC, a count of lines in the source code, including comment line, and (b) Logical LOC (LLOC) that counts the number

of ‘statements’ and for this reason is less sensitive to formatting and style conventions (Fenton & Bieman, 2014). The distinction between LOC and LLOC also applies to App Inventor. For example, the code snippet in Figure 5 has LOC = 2 and LLOC = 1: the second line of the *join* block is in a separate line for the sake of clarity, but it is just a logical continuation of the previous line (i.e., ‘set global greet to join(hola and textbox1.text)’).

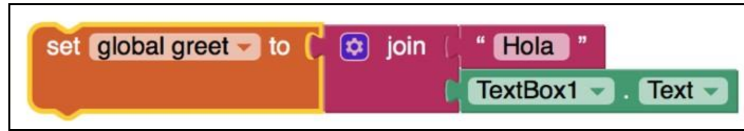

Figure 5. An App Inventor code snippet with LOC=2 and LLOC=1.

PROCESS ASSESSMENT

Our instructional strategy suggests a collaborative, iterative process in which participants can take control of the working pace, under the instructors’ supervision, with the time boundary of a relatively short bootcamp. Instructors do not dictate a process as it is but instead suggest several activities and recommendations to follow, as described widely in Section *Instructional Strategy*. For *process assessment*, we observe process-relevant traits focusing on XP practices (e.g., small releases and Iterations, refactoring, testing), namely, user stories and metaphor, small releases and iterations, refactoring/testing, teamwork, on-site customer, continuous integration, collective ownership, pair programming, coding standards. For instance, we observed if teams started using post-its (*user stories*) to guide the production process and decide when a prototype (*small releases* and *testing*) was ready for the *on-site customer* meeting.

CASE STUDIES

We set two case studies in the form of software development bootcamps with a segmented non-software population. Selected audiences were high school students (HS), and university postgraduate students of Arts (UPA).

High School students (HS). We held the bootcamp targeted to a class of high school seniors in Italy. We did not impose restrictions on the type of high school to create a more stimulating and multidisciplinary environment. The participants were 28 students (6 F and 22 M, aged 15-19) from a range of high school types: computer science (1), scientific (22), vocational (1), and non-vocational (4). Most of the participants had little or no previous knowledge of software development. They took part in the bootcamp to live the first software development experience and choose a future career. Thus, we focused on the process by which they arrived at the product (Steghöfer et al., 2016). The activities started on Monday afternoon and concluded on Friday afternoon, four hours per day.

University Postgraduate Students of Arts (UPA). The second case study took place in Mexico. Participants were a segmented population of seven adults (aged 28-47), studying a master’s degree program on Hypermedia Design in the Faculty of Fine Arts of a state-funded university. They needed to acquire a specific skill set that enables an Arts professional to mesh in an Engineering environment contributing with the visual aspects and user experience of software products (such as web pages and mobile applications). This class did not have previous knowledge of software development or CS background and included Graphic Designers, Industrial Designers, Visual Arts professionals. We used

the same training material as in the HS case study, translated to Spanish. Participants worked alone on personal projects (i.e., we omitted team-building games). Indeed, the size of the class was considerably smaller than the other case study, and each student had a specific idea that she/he wanted to develop.

RESULTS

This section illustrates the results of the case studies and demonstrates how to interpret the collected metrics to obtain an overall picture of the participants' performance. Moreover, we show how the selected metrics can help to find particular successful (or unsuccessful) cases as targets to be achieved (or avoided) using our instructional strategy (Gladwell, 2008). We use descriptive statistics to prevent concerns in the analysis that could be caused by the limited dimension of this quantitative data set (Wohlin et al., 2012; Zelkowitz & Wallace, 1998).

Product assessment (overall). In total, we collected 17 projects (10 HS and 7 UPA). The HS projects are larger (Figure 6), have higher complexity (Figure 7), more screens (which implies higher visual complexity), and higher TNC (Figure 8). It is important to recall that the HS participants worked in teams: the joint effort of several people, while introducing possible coordination and management issues, may have contributed to developing larger and more complex projects.

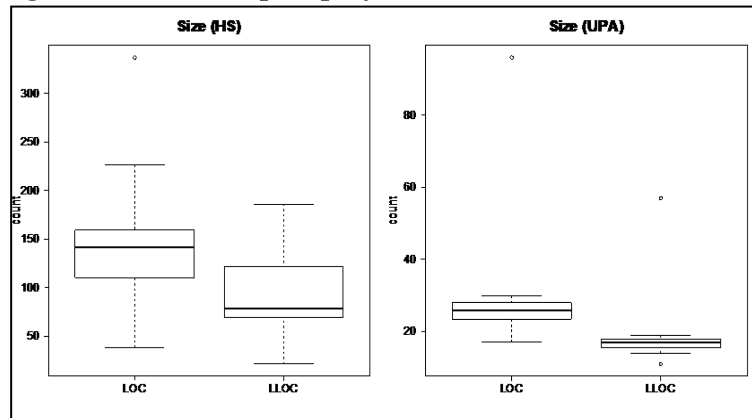

Figure 6. Project size in the two case studies.

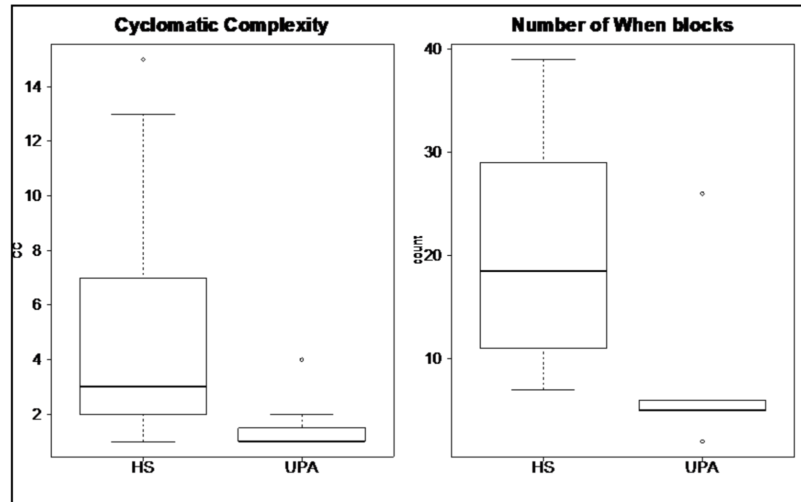

Figure 7. Project complexity in the two case studies.

The HS projects have a higher degree of variability (e.g., NOUB in Figure 8). The User Interface (UI) and media blocks are used a few times in both case studies, and some components (i.e., connectivity and social) have not been used at all, or just in one case. We will consider the corresponding projects as possible excellent products in the following analysis of individual projects.

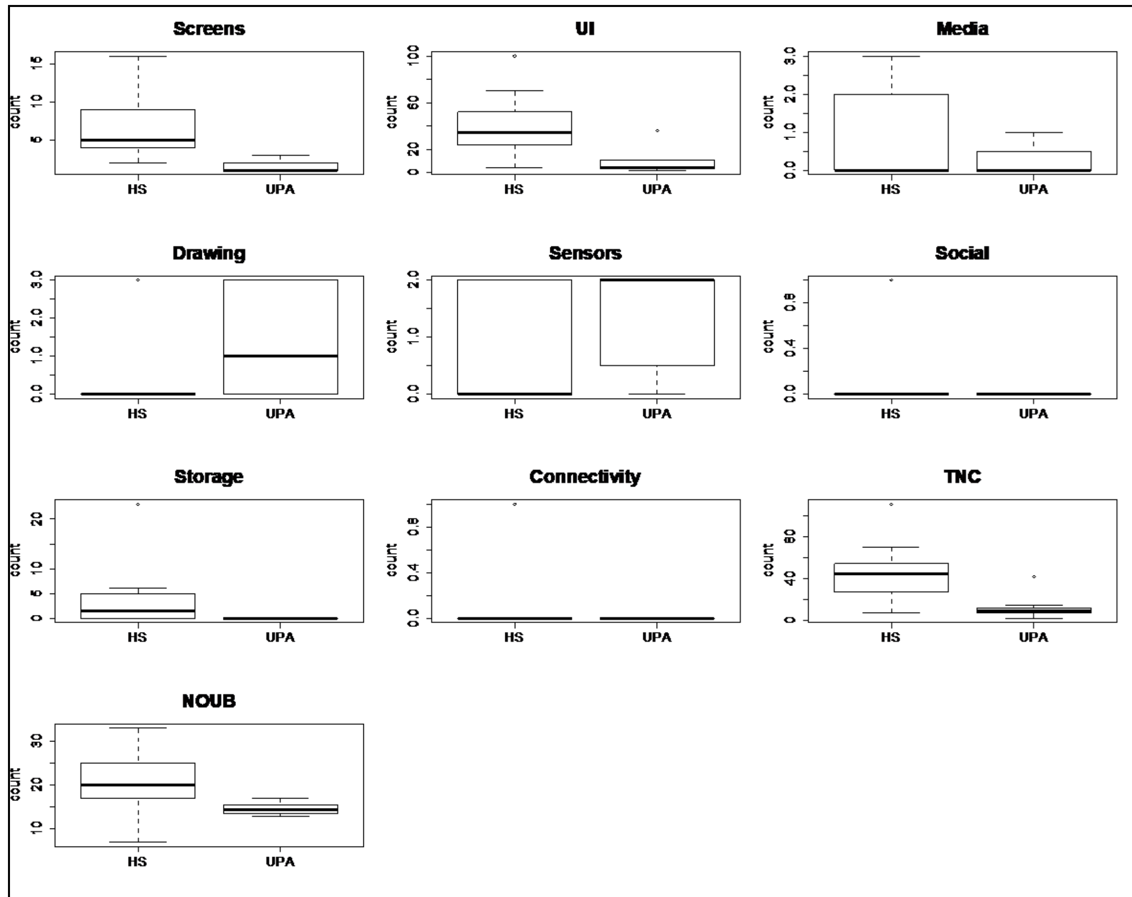

Figure 8. Components per type in the projects of the two case studies.

Loops and lists are used just in a few cases as well (see *computational concept blocks* in Figure 9). Moreover, HS projects contain a higher number of variables respect to the UPA ones (Figure 9).

As shown in Figure 10, some HS projects suffer from duplication, and long method smells. However, we should take into account that they are also more prominent in terms of size (Figure 6), which might increase the possibility of forgetting some blocks around. Moreover, UPA projects have lower complexity, which might help maintain a cleaner code and avoid long methods. We can observe the opposite behavior for the percentage of not-renamed components. UPA projects suffer from these smells more than the HS ones: the need for managing bigger and more complex projects might have pushed HS participants to start renaming components. These observations also explain the number of smells per project. Some HS projects concentrate a higher number of smells (60% of the project suffers from two smells, and 20% from three). None of the UPA projects suffers from three smells, and almost 60% suffer from one smell.

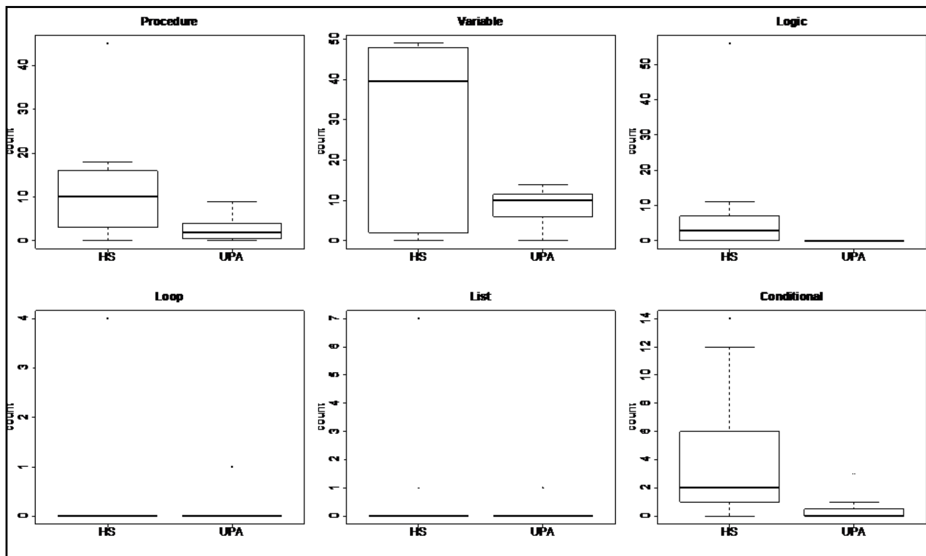

Figure 9. Computational concept blocks in the two case studies.

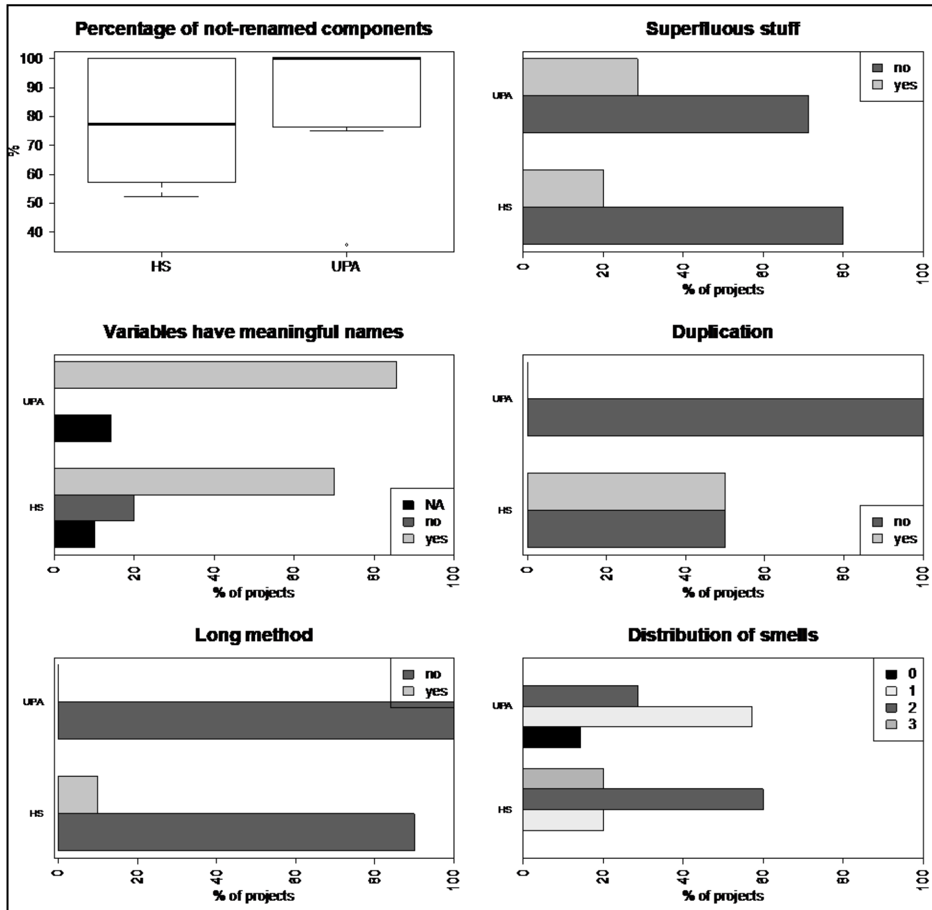

Figure 10. Code smells in the projects of the two case studies.

Product assessment (individual). Figure 11 highlights the presence of four projects with more peculiar characteristics than the other projects. The project *HS9* and *HS10* are larger (LLOC) respect to the others and, at the same time, present higher complexity (CC) and NOUB. Moreover, these projects contain computational concept blocks (i.e., social, connectivity, and logic) that are not present in other projects. The specific type of system has prompted the choice of these blocks: HS9 was a quiz and required to handle more conditions (i.e., logic), while HS10 included some features that required, for example, to send text messages. However, it is notable that these teams did not give up on adding these features, although it needed exploring the use of new blocks. The downside is the higher number of smells respect to the other projects. Since they had to handle size, complexity, and new blocks, these teams might have neglected other aspects (i.e., removing duplicate code and renaming some components).

Another interesting project is the *HS4*. Even if it is smaller and less complex than HS9 and HS10, the HS4 project has high NOUB, and it includes peculiar blocks (i.e., storage). The presence of smells is higher than the rest of the projects also in this case. A more balanced (thus ideal) situation can be found in *HS2*. This project shows average size and complexity; it includes a higher number of unique blocks (NOUB) and specific blocks (i.e., loops and lists). However, no smells are present.

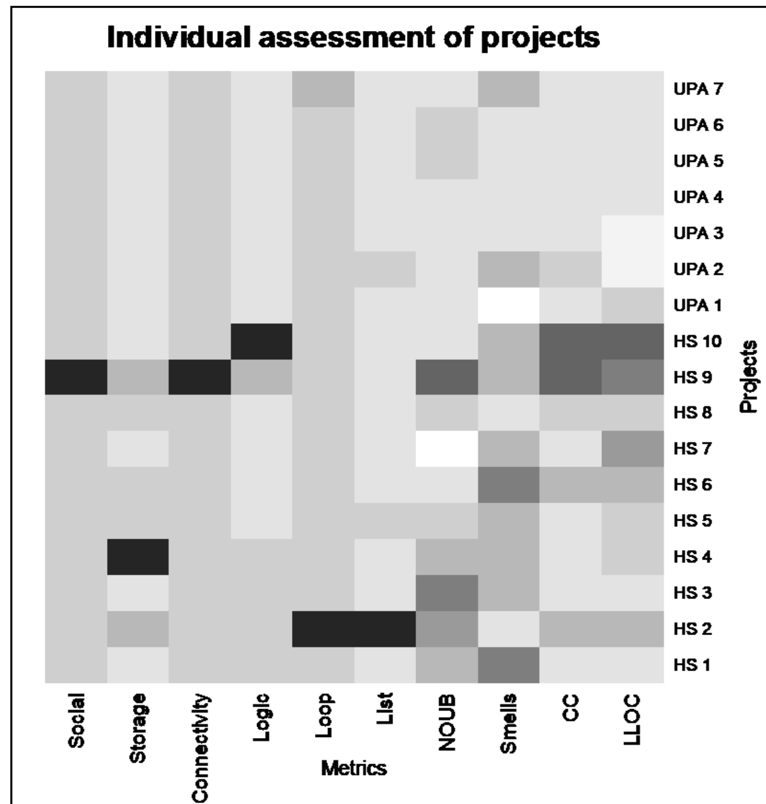

Figure 11. In this heatmap, lines and columns represent the 17 products and the set of metrics that show outliers in Figures 7-10, respectively. Larger values are represented by darker squares and smaller values by lighter squares.

PROCESS ASSESSMENT

Most of the case studies participants implemented a spontaneous software development process that used a collaborative and iterative approach and embodied many principles and values described in the suggested practices. When working in teams (HS case study), the participants defined a project goal and organized themselves in a way in which everybody could collaborate. Most of the participants did not have sufficient knowledge to propose themselves to lead the team; thus, the efforts were federated and distributed evenly among participants. In a few cases, previous experience, previous tool usage, or affinity with the topic facilitated that a specific student took the lead orchestrating the efforts and working personally on the most complex part of the project. For the observed case that involved individual work (i.e., UPA), goals were also set, and milestones for the day were established, so the process was also spontaneous and hand-crafted. Participants that acquired a more robust command of the tools served as technical aid to other individuals.

The limited amount of time allotted to develop the project prevents our instructional strategy to deepen inside each phase of the software development process; nonetheless, participants used all the resources at hand to walk the path from conceptual design to implementation. Intermediate deliverables (like drawings) place evidence of design activities that provide a notional idea about how the final product should look and feel. When participants reached the phase of software implementation, the BBPL provided an invaluable addition to developing software with little or no experience.

As detailed in Table 5, specific examples of process-relevant traits observed in the case studies are:

- A team implemented an effort of development and testing features, following involuntarily a continuous integration framework, in which each new feature developed was immediately tested to "ensure that the previous product still works".
- Several teams conducted independent online searches for features and tools to implement requirements that were not directly translated to block functionalities.
- Some participants attempted to create a single mobile application using different computers and different accounts. A team found out that merging the independent projects could represent an issue, and spontaneously converged to work in a pair programming-like setting.
- A team solved the same problem autonomously using an App Inventor-supported merger tool (MIT App Inventor, n.d.), which was searched and discovered independently by the team.

Considering that the UPA case study did not involve teamwork, it is relevant to list several traits that were observed in such context:

- The goals and topics of the developed applications were distinct and independent.
- Even though the work was individual, and the product delivered was submitted independently, the interaction between participants was strong.
- Participants were interacting, asking questions, and sharing experiences among themselves. When an individual sorted out a technical problem, that problem was shared with others.
- As a consequence, we could observe the rise of the figure of improvised "technical experts" who acted both as technical aid and as "peer reviewers".

In both the case studies, the validation efforts to assure product correctness were somewhat limited. Occasionally, groups did cross-checks with other teams, inviting them to use their app and provide comments about the features, functioning, and overall experience, or using the applications with the clear intent to crash it. However, no formal product assurance activity was required or recommended by instructors, other than looking for feedback and rework accordingly.

Table 5. Assessment of Agile Software Engineering Practices.

PRACTICE	OBSERVATIONS
User stories and Metaphor	Teams and individuals identified users, typically a target population of their same age and interests.

Small releases and Iterations	Software products had specific functionalities developed incrementally.
Refactoring, Testing	Some team members focused their efforts to find ways to crash the product, assuring as a consequence that the application worked as expected.
Teamwork	Each team identified the structure and amount of work to be done, created assignments and tasks, and distributed and executed them. Spontaneous technical experts provided support taking the role of peer reviewers.
On-site customer	Course instructors took the role of final customers, provided feedback, and refined requirements.
Continuous integration	Software products were developed and tested incrementally.
Collective ownership	Each team member created and explained a personal contribution. In individual work, ownership is intrinsically related to the creator, but the knowledge was collectively growth.
Pair programming	Since products were to be developed in a single project, teams typically were sitting down around a computer, so the roles of drivers and observers were quickly taken.
Coding standard	The block-programming tools enforced a single coding style that was mandatory.

ELABORATION OF A GENERAL STRATEGY FOR QUALITY ASSESSMENT

We found a significant (p-value < 0.05; adjusted R-squared = 0.90) linear relationship between the two metrics LOC and LLOC. Given the limited size of our sample, we cannot generalize this result, but this could suggest choosing LOC as a better metric for size because its calculation is usually easier (Khan et al., 2016) in respect to other metrics. However, considering that LOC is programmer- and language-dependent, and it does not take into consideration the code functionality, we interpret this result as an indicator that LLOC can be used instead of LOC. Indeed, in the specific case of App Inventor (and of BBPLs in general), LLOC might be preferable to recognize higher development effort. Consider a scenario of two programs P1 and P2 with the same LOC, while P2's LLOC is lower than in P1. This could mean that the P2's developer used a higher number of multi-line blocks (Figure 6), which represent more complex instructions. Therefore, to set up multi-line blocks, organize the parameters, and ensure that the block works, the developer needed more effort and more complex knowledge.

In our assessment framework, we consider the number of *when* blocks as an indicator of complexity specifically thought for BBPLs (Fronza et al., 2020). The regression model between When and Cyclomatic Complexity (CC) indicates a real relationship (p-value < 0.05), but the adjusted R-squared value is low, which tells us that the points are pretty scattered around the regression line. We cannot generalize this result due to the limited

sample size. However, since we do not need precise prediction, we believe we can still interpret this result as a further confirmation that the When metric can be used as a complexity metric instead of CC.

DISCUSSION

With the experience gained in the presented case studies, and after the in-depth analysis executed in the outcome products, we discuss several insights. By the results observed in separate groups, with the independence of placement, age, and backgrounds, we can argue that our instructional strategy, powered by flexible practices and ad-hoc development tools, facilitates the teaching/learning process of SE. Moreover, it sets on participants the capacity to understand the science behind working software, the effort that requires to produce it, and the practical implications that shape and polish such products. To better understand this assertion, we can decompose it (based on the central parts of our intervention, i.e., instructional strategy and assessment framework) and provide analysis and discussion on distinct fronts.

Instructional strategy. Our instructional strategy does not start directly from a hands-on approach. First, we direct the discussion to principles of logic, clear thinking, and organization that is required to understand the functioning of a generic software product. Using examples that are easily understood by the target populations, we underline the importance of abstracting a problem in terms of its inputs, steps, and outcomes, and with this mindset, develop a general approach to problem-solving. Due to time limitations, we do not claim that all the introduced principles are entirely understood and put into practice. However, these preparatory parts lay a foundation that eases the learning process when the structural and logical sections of the applications, and the associated development tools, are later explained.

Through teamwork and collaboration activities, participants walk the path of a creative and practical process that seldom is executed alone. Co-located and globalized software development teams usually work in close collaboration across or within different development phases. The application of our instructional strategy during the case studies, through games, activities, and team dynamics, demonstrates that teamwork and collaboration are a cornerstone of a successful product. The software development process is to be conducted once a team is focused on the development of the product, and its members have assembled a collaboration mechanism.

Due to the constraints of a minimal timeframe, it is challenging to evaluate with complete certainty how the participants embraced a comprehensive software development process. Nevertheless, we observed several traits that are common (and recommended) in software development: setting goals, breaking down high-level objectives in shorter activities, spontaneous planning, lookup of productivity tools, and informal testing.

Assessment framework. Our assessment framework capitalizes on the existing research to create a general strategy for App Inventor product quality assessment, from a SE perspective. In particular, the results of our case studies show how the framework can support the evaluation of the overall software quality capacity reached by the entire group of participants. We should bear in mind that this framework is a first approach to appraise the quality of the outcome software product. Even in its infancy, the framework has attributes and conditions that can be measured as software quality characteristics, and in

specific contexts, such attributes may be relevant. Moreover, the framework can be used to understand if outliers (Osborne & Overbay, 2004) represent particular successful or unsuccessful cases (Gladwell, 2008) that could be of great interest as targets to be achieved (or avoided) using our instructional strategy.

LIMITATIONS

We discuss the limitations based on a checklist by Runeson & Höst (2009). For *internal validity*, we acknowledge that we cannot exclude the possible effect of factors that we did not control. For example, we did not collect information on participants' usual performance at school/university; therefore, we may not exclude the possible effects of their inclination to study. Furthermore, we do not call for a particular participant's background or profile so that a group could be systematically replicated. Regarding *external validity*, it is well known that the results of case studies are difficult to generalize to other situations (Wohlin et al., 2012). Therefore, the results of our study can be extended to cases which have common characteristics and, hence, for which the findings are relevant.

Moreover, we acknowledge that the event proposed by our instructional strategy may attract students particularly interested in programming. Thus, further research that would look into the generalizability of these results in other situations is needed. To improve the reliability of our study, a detailed case study protocol was maintained and reviewed by two authors of this paper.

CONCLUSIONS

INITIAL QUESTIONS, REVISITED

We propose here a more insightful, directed discussion using as a starting point the questions left open in the first part of this work:

RQ1: How to adapt a Software Engineering instructional strategy to a diversified set of audiences with different backgrounds and needs during intensive project-based events?

With the lessons learned after our case studies, we can outline several modifications to adapt regular SE formation and obtain an instructional strategy for non-expert or non-specialized audiences:

- *Vocabulary*. It is of utmost importance to adapt vocabulary and narratives to the context of non-technical participants.
- *Customized examples*. It is required to find examples that are familiar to the target audience.

- *Teaching sequence*. The teaching sequence of the pure SE concepts cannot be traded or jeopardized: it is necessary to build a foundation of logical thinking, structured sequencing, and data abstractions before starting with coding practices.
- *A comprehensive view*. Other principles of SE, such as project management and team collaboration, cannot be forsaken or left apart: those are fundamental principles that complement the technical aspects and permit a healthy succession of tasks and organization of activities that translate into progress and delivery.

Table 6 highlights the importance of each component of our instructional strategy.

Table 6. Components of the instructional strategy: importance.

STRATEGY	WHY THE STRATEGY IS IMPORTANT
Manipulatable examples	Helps participants to have a clearer notion to understand the point of view of relevant actors of their products.
Focus on the problem-solving activity	Aids students to abstract tasks at a granular level; gives the notion of feasibility and feeling of accomplishment.
Alert without imposing	Creates awareness about implementing small changes with notable effects in the final product, discussing and sharing.
We are here to help	Empowers participants to raise questions, not holding back, but reaching out.
Block-Based Programming	Allows for the structure and practice of a programming language, with an approach that is friendly for novices.
Teamwork	An underlying principle of a professional experience that enables successful project development.
STRATEGY	WHY THE STRATEGY IS IMPORTANT
Marshmallow challenge	Cultivates a mindset of understanding the goal, assessing resources, and anticipating unexpected problems.
Tell me how you make toast	Fosters a goal-based vision, identification of simple solutions and abstraction in ordered steps.
Letters with our bodies	Consolidates teamwork, promotes agility on creating fast and efficient solutions.
User Persona and User Journey	Develops a sense of empathy upon the actors that are relevant or will interact with the product.
Point of View	Relates the human aspects of the User Persona and User Journey with the technical aspects of a User Story.

RQ2: How should we asses Software Quality, especially when considering non-conventional (e.g., block-based) development tools? What metrics apply in this case?

The applicability and usefulness of regular software metrics (e.g., McCabe, Halstead, LLOC) remain very valuable for block programming, as they permit the understanding of the software product in terms of size, complexity, and other SE relevant aspects. Depending on the type of software product and its functionality, other metrics might better

describe the software products developed using BBPLs. For instance, the number of features utilized by the application (e.g., number of sensors or number of antennas), the number of components visible in the user interface, the number of components not visible in the user interface.

RECOMMENDATIONS FOR EDUCATIONAL PRACTICE

Based on our results and lessons learned reported by instructors, in Table 7 we highlight a set of recommendations for educational practice concerning the understanding of the process and the products developed by inexperienced developers during the intensive project-based events.

Table 7. Recommendation for educational practice.

PRACTICE	LESSON LEARNED
Craftmanship	Instructors should influence students to incept the idea of crafting a product, from its conceptual design to the release of a working product.
Teamwork	There is value in identifying a high-level goal and break it down in clear objectives to be shared by team members. Instructors should be effective in facilitating the identification of the high-level goal and distributing objectives.
Technical Command	Technical accomplishment is king in the process of walking the line set by the bootcamp. Excellent command of the tools at hand enables instructors to facilitate the learning process of students and being effective in providing alternatives.
Accountability	Instructors should be careful about supervising that each team member responds to the team's needs by delivering to their commitments to avoid only one team member sustaining the workload.
Product Pride	Instructors should require that teams deliver a working product that goes beyond an anecdotal experience. Installed applications in personal mobile phones deliver a sense of accomplishment in participants.
Product quality	Instructors should suggest that teams pay attention to the quality of their code before increasing complexity and functionality.

In conclusion, we observed that teams with little or no background in software development could create a functional product using basic SE principles and ad-hoc development tools.

The case studies lay the foundation for interesting convergences: the proposed instructional strategy guarantees that all participants have sufficient knowledge of all software concepts through the resolution of examples and joint exercises. However, during the independent work, we noticed that the participants experienced problems working autonomously in their context.

The set of metrics included in the assessment framework represent a first approach to product evaluation for EUSE development; however, due to the small number of projects, it is difficult to find a single trend that we can associate to a particular profile of age or

technical experience. Critical trends like duplicated code or generic code smells also deliver insights on the relationship between functional code and top-quality code. Participants are encouraged to deliver working solutions, yet in the internal solution, the quality metrics are naturally slightly down compared to a professionally developed product.

From the software process point of view, the observation of games and group activities suggests that it is valuable to incorporate experiences that help participants to identify roles that are critical in SE and that could eventually be associated to typical roles such as Scrum Master, Product Owner, Software Tester, and others. Participants are empowered to define a path towards a successful product.

The present work offers a complementary and deeper view with respect to the scholarly literature produced to the date. In particular, in addition to the structural and instructional strategies already proposed, this paper works deeper in providing a framework to assess the quality of intermediate and final products. More work is needed to cover a profound implementation of an assessment framework for process and product that sheds light on the effectiveness of the instructional strategy and delivers a quantitative approach to determine courses of action, activities that should be continued, and practices that can be done differently.

Our instructional strategy lets participants identify a problem, select the most effective solution based on the introductory part, and finally, create the solution. Our instructional strategy collaborates to cultivate and benefit from software development skills and put them at the service of subjects of different fronts of their studies (in light that web technologies can be approached from the commercial, communication, visual design, and software development viewpoints).

Becker, K. (2015, 14-16 Oct.). Gamification: How to gamify learning and instruction. *Proceedings of the 2015 IEEE Games Entertainment Media Conference (GEM)*, Toronto, ON, Canada (p. 1-3). https://doi.org/10.1109/gem.2015.7377207

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. *Proceedings of the 2012 Annual Meeting of the American Educational Research Association (AERA'12)*, Vancouver, Canada (p. 1-25). Vancouver, Canada: AERA. http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf

Burnett M. (2009). What is end-user software engineering and why does it matter? In V. Pipek, M. B. Rosson, B. de Ruyter, & V. Wulf (Eds.), *End-User Development. IS-EUD 2009. Lecture Notes in Computer Science, 5435,* 15-26. Springer. https://doi.org/10.1007/978-3-642-00427-8_2

Burnett, M., & Myers, B. A. (2014). Future of end-user software engineering: beyond the silos. In *Proceedings of the on Future of Software Engineering* (pp. 201–211). https://doi.org/10.1145/2593882.2593896

Burning Glass Technologies. (2016). *Beyond point and click: The Expanding demand for coding skills.* https://www.burning-glass.com/research-project/coding-skills/

Champagne, J. (2016). *Are coding bootcamps worth it?* https://blog.capterra.com/are-coding-bootcamps-worth-it/

Chimalakonda, S., & Nori, K. V. (2013). What makes it hard to teach software engineering to end users? Some directions from adaptive and personalized learning. *Proceedings of the 2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*, San Francisco, CA (pp. 324–328). IEEE. https://doi.org/10.1109/cseet.2013.6595270

Costabile, M. F., Mussio, P., Parasiliti Provenza, L., & Piccinno, A. (2008). End users as unwitting software developers. In *Proceedings of the 4th International Workshop on End-User Software Engineering* (pp. 6–10). New York, NY, USA: ACM. https://doi.org/10.1145/1370847.1370849

Decker, A., Eiselt, K., & Voll, K. (2015). Understanding and improving the culture of hackathons: Think global hack local. *Proceedings of the 2015 Frontiers in Education Conference (FIE)* (pp. 1–8). IEEE. https://doi.org/10.1109/fie.2015.7344211

Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach*. CRC press.

Frieze, C., Hazzan, O., Blum, L., & Dias, M. B. (2006). Culture and environment as determinants of women's participation in computing: revealing the women-cs fit. *ACM SIGCSE bulletin, 38*, 22–26. https://doi.org/10.1145/1124706.1121351

Fronza, I., Corral, L., & Pahl, C. (2020). An approach to evaluate the complexity of block-based software product. *Informatics in Education*, *19*(1), 15-32. https://doi.org/10.15388/infedu.2020.02

Fronza, I., El Ioini, N., & Corral, L. (2016). Blending mobile programming and liberal education in a socialeconomic high school. *MOBILESoft '16: Proceedings of the International Conference on Mobile Software Engineering and Systems*, Austin, TX, USA, pp. 123-126. https://doi.org/10.1145/2897073.2897096

Fronza, I., El Ioini, N., & Corral, L. (2017). Teaching computational thinking using agile software engineering methods: A framework for middle schools. *ACM Transactions on Computing Education (TOCE)*, *17*(4),19. https://doi.org/10.1145/3055258

Fronza, I., El Ioini, N., Pahl, C., & Corral, L. (2019). Bringing the benefits of agile techniques inside the classroom: A practical guide. In D. Parsons & K. MacCallum (Eds.), *Agile and lean concepts for teaching and learning: Bringing methodologies from industry to the classroom* (pp. 133–152). Springer. https://doi.org/10.1007/978-98113-2751-3_7

Fronza, I., & Pahl, C. (2018a). End-user software engineering in K-12 by leveraging existing curricular activities. *ICSOFT 2018 - Proceedings of the 13th International Conference on Software Technologies*, pp. 249-255. https://doi.org/10.5220/0006846702830289

Fronza, I., & Pahl, C. (2018b) Envisioning a computational thinking assessment tool. *CEUR Workshop Proceedings*, Vol. 2190. http://ceur-ws.org/Vol-2190/TACKLE_2018_paper_2.pdf

Fronza, I., & Pahl, C. (2019) Teaching software engineering principles in non-vocational schools. *CSEDU 2019 - Proceedings of the 11th International Conference on Computer Supported Education*, 1, pp. 252-261. https://doi.org/10.5220/0007672702520261

Gama, K. (2019). Developing course projects in a hack day: An experience report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 388-394). https://doi.org/10.1145/3304221.3319777

Gama, K., Alencar Goncalves, B., & Alessio, P. (2018). Hackathons in the formal learning process. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (pp. 248–253). https://doi.org/10.1145/3197091.3197138

Gladwell, M. (2008). *Outliers: The story of success*. Hachette UK.

Grover, S. (2017). Tackling novice learners' naive conceptions in introductory programming. *Hello World, 2*.

Hermans, F., & Aivaloglou, E. (2016). Do code smells hamper novice programming? A controlled experiment on scratch programs. *Proceedings of 2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (pp. 1–10). https://doi.org/10.1109/icpc.2016.7503706

Hou, D., Jablonski, P., & Jacob, F. (2009). CNP: Towards an environment for the proactive management of copy-and-paste programming. In *Proceedings of 2009 IEEE 17th International Conference on Program Comprehension* (pp. 238–242). https://doi.org/10.1109/icpc.2009.5090049

Kamuto, M. B., & Langerman, J. J. (2017, May). Factors inhibiting the adoption of DevOps in large organisations: South African context. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology* (RTEICT) (p. 48-51). https://doi.org/10.1109/rteict.2017.8256556

Lara, M., & Lockwood, K. (2016) Hackathons as community-based learning: A case study. *TechTrends, 60,* 486495. https://doi.org/10.1007/s11528-016-0101-0

Kastl, P., Kiesmüller, U., & Romeike, R. (2016). Starting out with projects: Experiences with agile software development in high schools. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 60–65). https://doi.org/10.1145/2978249.2978257

Khan, A. A., Mahmood, A., Amralla, S. M., & Mirza, T. H. (2016). Comparison of software complexity metrics. *International Journal of Computing and Network Technology*, 4(1). https://doi.org/10.12785/ijcnt/040103

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., & Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys* (CSUR), *43*(3), 21. https://doi.org/10.1145/1922649.1922658

Liebenberg, J., Huisman, M., & Mentz, E. (2015). The relevance of software development education for students. *IEEE Transactions on Education*, *58*(4), 242-248. https://doi.org/10.1109/te.2014.2381599

Meerbaum-Salant, O., & Hazzan, O. (2010). An agile constructionist mentoring methodology for software projects in the high school. *ACM Transactions on Computing Education*, *9*(4), n4. https://doi.org/10.1145/1656255.1656259

Millis, B. J., & Cottell, P. G., Jr. (1997). *Cooperative learning for higher education faculty*. Series on higher education. Oryx Press.

MIT App Inventor. (n.d.). *AI2 Project Merger Tool: Combine two App Inventor projects into one*. http://appinventor.mit.edu/explore/resources/ai2-project-merger.html

MIT App Inventor. (2020, June 9). *MIT App Inventor stats* http://ai2.appinventor.mit.edu/stats

Morelli, R., de Lanerolle, T., Lake, P., Limardo, N., Tamotsu, E., & Uche, C. (2011). Can android app inventor bring computational thinking to K-12? In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM. https://doi.org/10.1145/2157136.2157437

Moreno-Leon, J., Robles, G., & Román-Gonzalez, M. (2015). Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED-Revista de Educacion a Distancia*, *46*, 1–23. https://doi.org/10.6018/red/46/10

Moreno-Léon, J., Robles, G., & Román-González, M. (2016). Comparing computational thinking development assessment scores with software complexity metrics. In *Global Engineering Education Conference* (EDUCON), 2016 IEEE (pp. 1040–1045). https://doi.org/10.1109/educon.2016.7474681

Nichols, M., Cator, K., & Torres, M. (2016). *Challenge based learner user guide*. Digital Promise.

Oakley, B., Felder, R. M., Brent, R., & Elhajj, I. (2004). Turning student groups into effective teams. *Journal of Student-Centered Learning*, *2*(1), 9–34.

Osborne, J. W., & Overbay, A. (2004). The power of outliers (and why researchers should always check for them). *Practical Assessment, Research & Evaluation*, *9*(6), 1–12. https://doi.org/10.7275/qf69-7k43

Ota, G., Morimoto, Y., & Kato, H. (2016). Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),* (pp. 238–239). https://doi.org/10.1109/vlhcc.2016.7739695

Paternò, F. (2013). End user development: Survey of an emerging field for empowering people. *International Scholarly Research Notices*, *2013*, Article 532659. https://doi.org/10.1155/2013/532659

Porras, J., Ikonen, J., Heikkinen, K., Koskinen, K., & Ikonen, L. (2005). Better programming skills through code camp approach. In *16th EAEEIE Annual Conference on Innovation in Education for Electrical and Information Engineering* (pp. 6–8).

Porras, J., Khakurel, J., Ikonen, J., Happonen, A., Knutas, A., Herala, A., & Drögehorn, O. (2018). Hackathons in software engineering education-lessons learned from a decade of events. In *Proceedings of the 2nd International Workshop on Software Engineering Education for Millennials* (pp. 40-47). ACM. https://doi.org/10.1145/3194779.3194783

Romeike, R. & Göttel, T. (2012). Agile projects in high school computing education: Emphasizing a learners' perspective. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education*, pages 48–57. ACM. https://doi.org/10.1145/2481449.2481461

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131. https://doi.org/10.1007/s10664-008-9102-8

Sammut-Bonnici, T., & McGee, J. (2015). Case study. In C.L. Cooper, J. McGee, & T. Sammut-Bonnici (Eds.), *Wiley Encyclopedia of Management (Vol 12)*. John Wiley & Sons. https://doi.org/10.1002/9781118785317.weom120012

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers.
2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05 (pp. 207–214). https://doi.org/10.1109/vlhcc.2005.34

Scheubrein, R. (2003). Elements of end-user software engineering. *INFORMS Transactions. on Education*, *4*(1), 37–47. https://doi.org/10.1287/ited.4.1.37

Shaw, M. (2000). Software engineering education: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 371–380). https://doi.org/10.1145/336512.336592

Steghöfer, J.-P., Knauss, E., Alégroth, E., Hammouda, I., Burden, H., & Ericsson, M. (2016). Teaching agile: Addressing the conflict between project delivery and application of agile methods. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 303–312). https://doi.org/10.1145/2889160.2889181

Uys, W. F. (2019). Hackathons as a formal teaching approach in information systems capstone courses. In *Annual Conference of the Southern African Computer Lecturers' Association 2019* (pp. 79-95). Springer. https://doi.org/10.1007/978-3-030-35629-3_6

Waite, J. (2017). Smelly code. Do we pass on best practice when we teach block-based programming to primary school pupils? *Hello World*, 3.

Warren, N. (2018). *What is citizen development? And, how can you govern citizen developers more effectively?*
https://www.outsystems.com/blog/posts/citizen-developer/

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

Wolber, D., Abelson, H., Spertus, E., & Looney, L. (2011). *App Inventor*. O'Reilly Media.

Wujec, T. (2010). *Build a tower, build a team* [Video]. TED Conferences. https://www.ted.com/talks/tom_wujec_build_a_tower_build_a_team

Wujec, T. (2013). *Got a wicked problem? First, tell me how you make toast* [Video]. TED Conferences.
https://www.ted.com/talks/tom_wujec_got_a_wicked_problem_first_tell_me_how_you_make_toast

Xie, B., & Abelson, H. (2016). Skill progression in MIT App Inventor. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 213–217). https://doi.org/10.1109/vlhcc.2016.7739687

Xie, B., Shabir, I., & Abelson, H. (2015). *Measuring the programmatic sophistication of app inventor projects grouped by functionality*. http://web.mit.edu/bxie/www/thesis.pdf

Ye, Y., & Fischer, G. (2007). Designing for participation in socio-technical software systems. In C. Stephanidis (Ed.), *Universal Access in Human Computer Interaction. Coping with Diversity. UAHCI 2007. Lecture Notes in Computer Science, 4554,* 312-321. Springer. https://doi.org/10.1007/978-3-540-73279-2_35

Zelkowitz, M. V., & Wallace, D. R. (1998). Experimental models for validating technology. *Computer*, *31*(5), 23–
31. https://doi.org/10.1109/2.675630