Informatics in Education, 2018, Vol. 17, No. 1, 117–150

DOI: 10.15388/infedu.2018.08

CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs

Christiane Gresse von WANGENHEIM[1],
Jean C. R. HAUCK[1], Matheus Faustino DEMETRIO[1],
Rafael PELLE[1], Nathalia da CRUZ ALVES[1],
Heliziane BARBOSA[2], Luiz Felipe AZEVEDO[2]

[1]*Department of Informatics and Statistics, Federal University of Santa Catarina Florianópolis/SC, Brazil*
[2]*Department of Graphic Expression, Federal University of Santa Catarina Florianópolis/SC, Brazil*
e-mail: {c.wangenheim, jean.hauck}@ufsc.br, {matheus.demetrio, rafaelpelle}@grad.ufsc.br, nathalia.alves@posgrad.ufsc.br, {heliziane.barbosa, felipe.azevedo}@grad.ufsc.br

Received: November 2017

Abstract. The development of computational thinking is a major topic in K-12 education. Many of these experiences focus on teaching programming using block-based languages. As part of these activities, it is important for students to receive feedback on their assignments. Yet, in practice it may be difficult to provide personalized, objective and consistent feedback. In this context, automatic assessment and grading has become important. While there exist diverse graders for text-based languages, support for block-based programming languages is still scarce. This article presents CodeMaster, a free web application that in a problem-based learning context allows to automatically assess and grade projects programmed with App Inventor and Snap!. It uses a rubric measuring computational thinking based on a static code analysis. Students can use the tool to get feedback to encourage them to improve their programming competencies. It can also be used by teachers for assessing whole classes easing their workload.

Keywords: computational thinking, programming, assessment, grading, app inventor, Snap!

1. Introduction

Computational Thinking (CT) is a competence that involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science (Wing, 2006). It is considered a key competence for today's generation of students in a world that is heavily influenced by computing principles (Wing, 2006). Therefore, teaching computational thinking has been a focus

of worldwide efforts of computing K-12 education (Grover and Pea, 2013) (Kafai and Burke, 2013) (Resnick *et al.*, 2009). Many of these initiatives focus on teaching programming, which is not only a fundamental part of computing, but also a key tool for supporting the cognitive tasks involved in computational thinking (Grover and Pea, 2013). Programming in K-12 is typically taught using visual block-based programming languages, such as Scratch (`https://scratch.mit.edu`), BYOB/Snap! (`http://snap.berkeley.edu`) or App Inventor (`http://appinventor.mit.edu/explore`) (Lye and Koh, 2014). Block-based programming languages encourage and motivate to learn programming concepts reducing the cognitive load by allowing to focus on the logic and structures involved in programming rather than requiring to learn the complex syntax of text-based programming languages (Kelleher and Pausch, 2005)(Maiorana *et al.*, 2015)(Grover *et al.*, 2015). Furthermore, they allow students to enact computational practices more easily as the outcomes of their programming can be viewed immediately in the form of animated objects, games or apps. This enables students to acquire computational problem-solving practices more easily adopting an engineering design cycle (Lye and Koh, 2014). Thus, many instructional units include mainly hands-on programming activities to allow students to practice and explore computing concepts effectively as part of the learning process (Lye and Koh, 2014) (Grover and Pea, 2013) (Wing, 2006). This includes diverse types of programming activities, including closed-ended problems for which a correct solution exists, such as, e.g., programming exercises from Hour of Code (`https://hourofcode.com`) (Kindborg and Scholz, 2006). Many computational thinking activities also focus on creating solutions to real-world problems, where solutions are software artifacts, such as games/animations on interdisciplinary topics or mobile apps to solve a problem in the community (Monroy-Hernández and Resnick, 2008) (Fee and Holland-Minkley, 2010). In such constructionist-based problem-based learning environments, student learning centers on complex ill-structured, open-ended problems, lacking explicit parameters without a unique correct answer or solution path (Lye and Koh, 2014) (Fortus *et al.*, 2004) (Gijselaers, 1996) (Shelton and Smith, 1998) (Simon, 1983). Educationally sound, especially such ill-structured problems engage students in deep problem-solving and critical thinking (Fee and Holland-Minkley, 2010) (Gallagher, 1997).

A crucial element in the learning process is assessment and feedback (Hattie and Timperley, 2007) (Shute, 2008) (Black and Wiliam, 1998). Assessment guides student learning and provides feedback for both the student and the teacher (Ihantola *et al.*, 2010). For effective learning, students need to know their level of performance on a task, how their own performance relates to good performance and what to do to close the gap between those (Sadler, 1989). Formative feedback, thus, consists of information communicated to the student with the intention to modify her/his thinking or behavior for the purpose of improving learning (Shute, 2008). Summative assessment aims to provide students with information concerning what they learned and how well they mastered the course concepts (Merrill *et al.*, 1992) (Keuning *et al.*, 2016). Assessment also helps teachers to determine the extent to which the learning goals are being met (Ihantola *et al.*, 2010).

Despite the many efforts aimed at dealing with the issue of CT assessment (Grover and Pea, 2013) (Grover *et al.*, 2015), so far there is no consensus on strategies for assessing CT concepts (Brennan and Resnick, 2012) (Grover *et al.*, 2014). Assessment of CT is particularly complex due the abstract nature of the construct being measured (Yadav *et al.*, 2015). Several authors have proposed different approaches and frameworks to try to address the assessment of this competence in different ways, including the assessment of student-created software artifacts as one way among multiple means of assessment (Brennan and Resnick, 2012). The assessment of a software program may cover diverse quality aspects such as correctness, complexity, reliability, conformity to coding standards, etc.

Yet, a challenge is the assessment of complex, ill-structured activities as part of problem-based learning. Whereas the assessment of closed-ended, well-structured programming assignments is straight-forward since there is a single correct answer to which the student-created programs can be compared (Funke, 2012), assessing complex, ill-structured problems for which no single correct solution exist is more challenging (Eseryel *et al.*, 2013) (Guindon, 1988). In this context, authentic assessment based on the created outcomes seems to be an appropriate means (Torrance, 1995; Ward and Lee, 2002). Thus, program assessment is based on the assumption that certain measurable attributes can be extracted from the program, evaluating whether the students-created programs show that they have learned what they were expected using rubrics. Rubrics use descriptive measures to separate levels of performance on a given task by delineating the various criteria associated with learning activities (Whittaker *et al.*, 2001) (McCauley, 2003). Grades are determined by converting rubric scores to grades. Thus, in this case the created outcome is assessed and a performance level for each criterion is assigned as well as a grade in order to provide instructional feedback.

Another issue that complicates the assessment of CT in K-12 education in practice is that the manual assessment of programming assignments requires substantial resources with respect to time and people, which may also hinder scalability of computing education to larger number of students (Eseryel *et al.*, 2013) (Romli *et al.*, 2010) (Ala-Mutka, 2005). Furthermore, as, due to a critical shortage of K-12 computing teachers (Grover *et al.*, 2015), many non-computing teachers introduce computing education in an interdisciplinary way into their classes, they face challenges also with respect to assessment as they do not necessarily have a computer science background (DeLuca and Klinger, 2010) (Popham, 2009) (Cateté *et al.*, 2016). This may further complicate the situation leaving the manual assessment error prone due to several reasons such as inconsistency, fatigue, or favoritism (Zen *et al.*, 2011).

In this context, the adoption of automatic assessment approaches can be beneficial by easing the teacher's workload leaving more time for other activities with students (Ala-Mutka and Järvinen, 2004). It can also help to ensure consistency and accuracy of assessment results as well as eliminating bias (Romli *et al.*, 2010). For students, it can provide immediate feedback on their programs, allowing them to make progress without a teacher by their side (Douce *et al.*, 2005) (Wilcox, 2016) (Yaday *et al.*, 2015). Thus, automating the assessment can beneficial for both students and teachers, improv-

ing computing education, even more in the context of online learning and MOOCs (Vujosevic-Janicica *et al.*, 2013).

As a result, automated grading and assessment tools for programming exercises are already in use in many ways in higher education (Ala-Mutka, 2005) (Douce *et al.*, 2005). The most widespread approach currently used for the automatic assessment of programs is through dynamic code analysis (Douce *et al.*, 2005). Dynamic approaches focus on the execution of the program through a set of predefined test cases, comparing the generated output with the expected output (provided by test cases). The main aim of dynamic analysis is to uncover execution errors and help to evaluate the correctness of a program. An alternative is static code analysis, the process of examining source code without executing the program. It is used for programming style assessment, syntax and semantics errors detection, software metrics, structural or non-structural similarity analysis, keyword detection or plagiarism detection, etc. (Fonte *et al.*, 2013). And, although there exist already a variety of automated systems for assessing programs, the majority of the systems is targeted only for text-based programming languages such as Java, C/C++, etc. (Ihantola *et al.*, 2010). There still is a lack of tools that support the evaluation of block-based programs assessing the development of CT, with only few exceptions mostly assessing Scratch projects such as Dr. Scratch (Moreno-León and Robles, 2015a) or Ninja Code Village (Ota *et al.*, 2016). These tools adopt static code analysis to measure the software complexity based on the kind and number of blocks used in the program quantifying CT concepts and practices such as abstraction, logic, control flow, etc. Allowing the assessment of ill-structured, open-ended programming activities, they provide instructional feedback based on a rubric. Tools for assessing programming projects in other block-based languages such as Snap! Autograder (Ball and Garcia, 2016) or App Inventor Quizly (Maiorana *et al.*, 2015), adopting a dynamic analysis approach only allow the assessment of closed-ended problems.

Thus, in this respect we present CodeMaster, a free web tool that analyzes App Inventor or Snap! programs to offer feedback to teachers and students assigning a CT score to programming projects. Students can use this feedback to improve their programs and their programming competencies. The automated assessment can be used as part of the learning process for formative, summative and/or informal assessment of CT competencies, which may be further enhanced by teachers revising and completing the feedback manually with respect to further important criteria such as creativity and innovation.

2. Background

2.1. *Block-Based Programming Environments*

Block-based programming environments are a variety of visual programming languages that leverage a primitives-as-puzzle pieces metaphor (Weintrop and Wilensky, 2015). In such environments, students can assemble programs by snapping together instruc-

tion blocks and receiving immediate feedback on if a given construction is valid. The construction space in which the blocks are used to program often also provides a visual execution space and/or a live testing environment in which the created programs can be tested throughout the development process. This supports an iterative development cycle allowing the student to easily explore and get immediate feedback on their programming (Wolber *et al.*, 2014).

In recent years, there has been a proliferation of block-based programming environments with the growing introduction of computing education in K-12. Well known block-based programming environments, such as Scratch (`https://scratch.mit.edu`), provide students with exploratory spaces designed to support creative activities creating animations or games. And, although Scratch is being currently one of the most popular environments, other environments such as App Inventor are also increasingly adopted, enabling the development of mobile applications as well as Snap! as an open-source alternative to Scratch providing also higher level programming concepts.

2.1.1. *App Inventor*

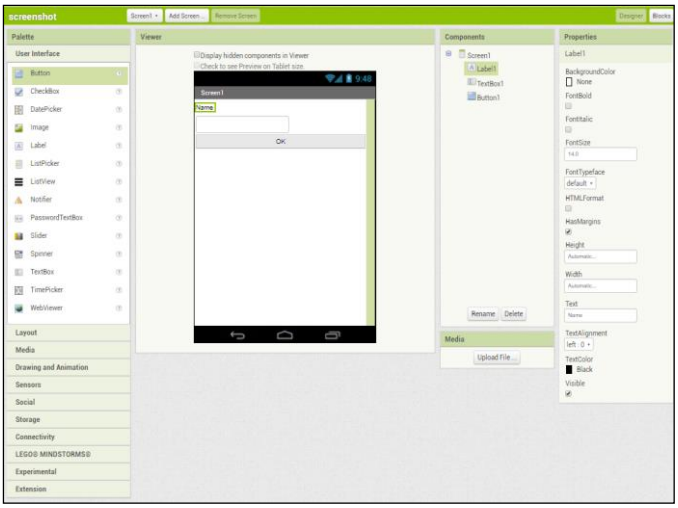
App Inventor (`http://appinventor.mit.edu`) is an open-source block-based programming environment for creating mobile applications for Android devices. It is an online browser-based programming environment using a drag-and-drop editor. It has been originally provided by Google and is now maintained by the Massachusetts Institute of Technology (MIT). The current version is App Inventor 2, retiring App Inventor Classic in 2015.

With App Inventor, a mobile app can be created in two stages. First, the user interface components (e.g., buttons, labels,) are configured in the Component Designer (Fig. 1). The Component Designer also allows to specify non-visual components such as sensors, social and media components accessing phones features and/or other apps.

In a second stage, the behavior of the app is specified by connecting visual blocks that correspond to abstract syntax tree nodes in traditional programming languages. Blocks represent events, conditions, or actions for a particular app component (e.g., button pressed, take a picture with the camera) while others represent standard programming concepts (e.g., conditionals, loops, procedures, etc.) (Turbak *et al.*, 2017). The app's behavior is defined in the Blocks Editor (Fig. 1b).

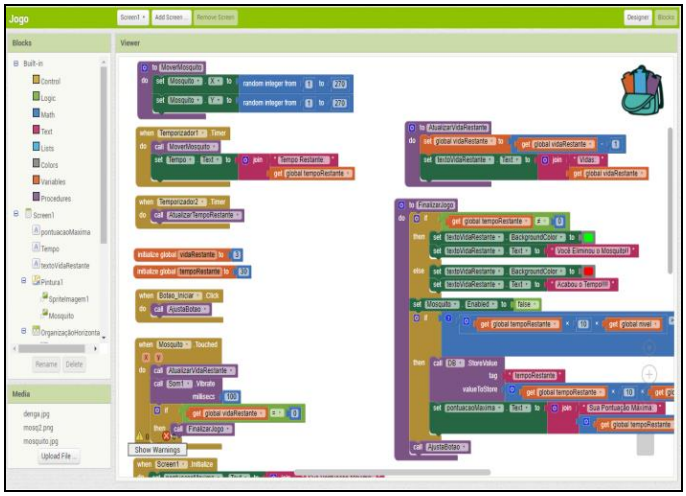
App Inventor allows to visualize the behavioral and/or visual changes of the application through the mobile application App Inventor Companion, which runs the app being developed in real-time on an Android device during development.

App Inventor project source code files are automatically saved in the cloud, but can also be exported as .aia files. An .aia file is a compressed collection of files that includes a project properties file, media files used by the app, and, for each screen in the app, two files are generated: a .bky file and .scm file. The .bky file encapsulates an xml structure with all the programming blocks used in the application logic, and the .scm file encapsulates a json structure that contains all the visual components used in the app (Mustafaraj *et al.*, 2017).



Designer
User interface
Layout
Media
Drawing and Animation
Sensors
Social
Storage
Connectivity
Lego Mindstorms
Experimental
Extensions

(a)



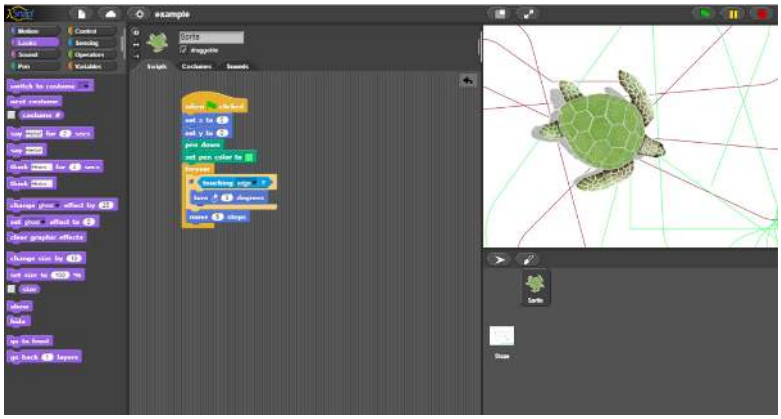
Blocks
Control
Logic
Math
Text
Lists
Colors
Variables
Procedures

(b)

Fig. 1. (a) Component Designer and block categories, (b) Blocks Editor and block categories

2.1.2. *Snap!*

Snap! (`http://snap.berkeley.edu`) is an open-source, block-based programming language that allows to create interactive animations, games, etc. Snap! 4.0 is an online browser-based programming environment using a drag-and-drop editor. Snap! 4.0 and its predecessor BYOB were developed for Linux, OS X or Windows (Harvey &



Blocks
Control
Motion
Looks
Sensing
Sound
Operators
Variables
Pen

Fig. 2. Snap! Blocks Editor and block categories.

Mönig, 2017) and have been used to teach introductory courses in computer science (CS) for non-CS-major students at the University of California. Snap! was inspired by Scratch, but also targets both novice and more advanced students by including and expanding Scratch's features including concepts such as first class functions or procedures ("lambda calculus"), first class lists (including lists of lists), first class sprites (prototype-oriented instance-based classless programming), nestable sprites and codification of Snap! programs to mainstream languages such as Python, JavaScript, C, etc. (Harvey *et al.,* 2012).

Snap! Projects are programmed in the visual editor (Fig. 2). Blocks are grouped into palettes, such as control, motion, looks, etc. A Snap! program consists of one or more scripts, each of which is made of blocks, being assembled by dragging blocks from a palette into the scripting area. The created programs can be executed directly in the stage section of the editor (Harvey and Mönig, 2017).

Snap! project source code files can be saved locally or in the Snap! cloud (requiring an account), but can also be exported as .xml files. The .xml file contains the code blocks and other elements used in the project, including all media such as images and sounds (in hexadecimal format).

2.2. *Assessment and Grading*

As with any pedagogic approach, it is important to align learning outcomes, teaching and learning activities and assessment, particularly when the intention is to encourage deep, rather than surface approaches to learning (Biggs, 2003). Thus, for assessing problem-based learning, authentic assessment seems a more appropriate means to assess learning compared to traditional assessments such as norm-reference and standardized testing that assesses recall of factual content knowledge (Torrance, 1995)(Ward and Lee, 2002). Authentic assessment measures performance based on the created outcomes or observed

performance in learning activities that encourage students to use higher-order thinking skills. There exist diverse types of authentic assessments in the context of problem-based learning such as performance assessments, portfolio assessment, interviews, self-assessments etc. (Hart, 1994) (Brennan and Resnick, 2012). Specifically, performance assessments measure students' ability to apply acquired competences in ill-structured contexts and working collaboratively to solve complex problems (Wiggins, 1993). Performance assessments typically require students to complete a complex task, such as programming a software artifact.

In performance assessments, in order to evaluate whether the work produced by students shows that they have learned what they were expected to learn, often rubrics are used. Rubrics use descriptive measures to separate levels of performance on the achievement of learning outcomes by delineating the various criteria associated with learning activities, and indicators describing each level to rate student performance (Whittaker *et al.*, 2001) (McCauley, 2003). When used in order to assess programming activities, such a rubric typically maps a score to the ability of the student to develop a software artifact (Srikant and Aggarwal, 2013) indirectly inferring the achievement of CT competencies. Rubrics usually are represented as a 2D grid that describes (Becker, 2003) (McCauley, 2003):

- Criteria: identifying the trait, feature or dimension to be measured.
- Rating scale: representing various levels of performance that can be defined using either quantitative (i.e., numerical) or qualitative (i.e., descriptive) labels for how a particular level of achievement is to be scored.
  - Levels of performance: describe the levels specifying behaviors that demonstrate performance at each achievement level.
  - Scores: a system of numbers or values used to rate each criterion and that are combined with levels of performance.
- Descriptors: describing for each criterion what performance at a particular performance level looks like.

So far there exist very few rubrics for assessing CT and/or programming competencies in the context of K-12 education. Some of them focus on closed-ended programming activities using indicators related to the evaluation of program correctness and efficiency (Srikant and Aggarwal, 2014) (Smith and Cordova, 2005), programming style (Smith and Cordova, 2005) and/or aesthetics and creativity, including not only the program itself but also documentation (Becker, 2003) (Smith and Cordova, 2005). Others are defined for a manual assessment of programming projects (Eugene *et al.*, 2016) (Becker, 2003) not supporting automated assessments. On the other hand, Moreno-León *et al.* define a rubric to calculate a CT score based on the analysis of Scratch programs automated through the Dr. Scratch tool (Moreno-León *et al.*, 2017) (Moreno-León and Robles, 2015b). The rubric is based on the framework for assessing the development of computational thinking proposed by Brennan & Resnick (2012), covering the key dimensions of computational concepts (concepts students engage with as they program, such as logical thinking, data representation, user interactivity, flow control, parallelism and synchronization) and computational practices (practices students develop as they engage with the concepts, focusing on abstraction). Specifically for App Inventor

projects, Sherman *et al.* developed a rubric to assess mobile computational thinking, including programming aspects typically related to computational thinking as represented by the computing practice & programming strand of the CSTA K12 standard as well as related concepts that are present in mobile computing with App Inventor Classic, e.g., screen design, eventbased programming, locationawareness, and persistent and shared data (Sherman and Martin, 2015) (Sherman *et al.*, 2014).

Rubrics also provide an objective basis for grading by converting rubric scores to grades. A grade is an indication of the level of performance reflected by a particular score of an assessment. Grades can be assigned as letters (generally A through F), as a range (for example, 0 to 10), as a percentage of a total number of questions answered correctly, or as a number out of a possible total (for example, 20 out of 100).

2.3. *Automatic Code Analysis for Assessment and Grading*

Indicating student performance, programming projects can be assessed with respect to diverse quality factors and characteristics. These indicators are measured automatically typically either through a dynamic or a static code analysis (Koyya *et al.*, 2013) (Ala-Mutka, 2005). Static analysis is the process of examining source code without executing the program. It is used to analyze static features like coding style, software metrics, programming errors (e.g., dead code), design, and special features related to program structure, as well as diagram analysis, keyword detection, and plagiarism detection (Ala-Mutka, 2005). Dynamic analysis is the process of executing the code and comparing the generated output to the control output as specified in a test case (Benford *et al.*, 1995). The main aim of dynamic analysis is to uncover execution errors and to evaluate the correctness of a program, as well as to evaluate efficiency and/or students' testing competencies.

Automated grading has been explored mostly for closed-ended well-structured assignments, for which a correct answer is known, as in this case it is easy to compare the student's program to the correct solution in order to evaluate correctness (Forsythe and Wirth, 1965) (Al-Matka, 2005). However, as dynamic and some static code analysis approaches depend on their ability to recognize and discriminate between correct and incorrect program behavior, they may not be viable solutions in problem-based learning contexts. Thus, for assessing open-ended ill-structured assignments for which no unique correct solution exists, typically static code analysis is used, focusing typically on the analysis of keywords, programming style, software metrics and/or plagiarism by counting the types and number of blocks used in a program.

3. Research Methodology

This article presents an exploratory research on the automation of the assessment and grading of ill-structured programming assignments in K-12 computing education. Therefore, we adopt a multi-method research strategy (Fig. 3).

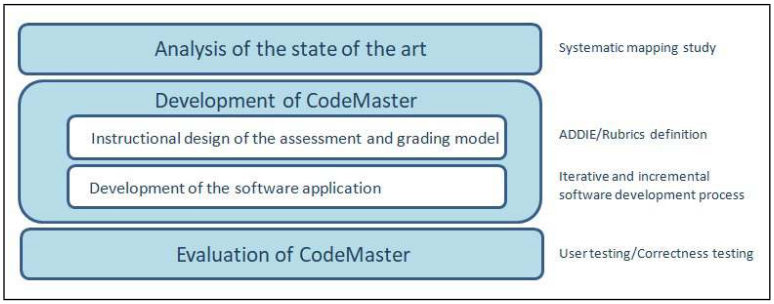


Fig. 3. Overview on the research method.

Analysis of the state of the art. In order to provide an overview on the current state of the art on the automatic assessment and grading of block-based programming assignments, we performed a systematic mapping study following the procedure proposed by Petersen *et al.* (2015) and Kitchenham *et al.* (2011). In the definition phase, the research questions and the review protocol were defined as well as the data sources, search strategy, search strings and inclusion/exclusion criteria. The execution phase was carried out based on the review protocol conducting the search in the specified repositories. The initial search results were analyzed with respect to their relevancy applying the inclusion/exclusion criteria. Once identified the relevant studies, data with respect to the research questions was extracted. Based on the extracted data the encountered studies were analyzed and results synthesized. The detailed results of this review are presented by Alves *et al.* (2017).

Instructional design of the assessment and grading. Following the ADDIE model (Branch, 2010), we initially performed a context analysis, characterizing the target audience and environment, as well as, identifying typical learning objectives and instructional strategies for K-12 computing education. In accordance with the identified context, we developed an assessment model by defining rubrics following the procedure proposed by Goodrich (1996). We first analyzed existing frameworks and rubrics for this specific context. In accordance to these existing models and the specific features of both block-based programming environments (App Inventor and Snap!), we identified assessment criteria. Then, we specified the levels of performance descriptors with respect to the identified learning objectives. Based on the defined rubrics, we defined a grading system in alignment with the generic grading system typically adopted in Brazil.

Development of the CodeMaster software application. Adopting an iterative and incremental software development approach (Larman and Basili, 2003), we analyzed the requirements based on the context analysis and the defined assessment and grading model. We, then, iteratively and incrementally developed the software tool by first specifying the use cases together with the interface design. In the next iterations, we implemented the system, starting with the implementation of a first prototype of the analysis & grader module, responsible for evaluating the projects. We, then, developed the presentation module, responsible for the user interaction and the persistence layer. Then, integration

tests were performed and based on the results, necessary corrections and improvements were implemented and tested.

Evaluation of CodeMaster. In order to evaluate the quality of the CodeMaster prototype, we performed a preliminary evaluation using the *Goal Question Metric* (GQM) approach (Basili *et al.*, 1994) to define the purpose of the evaluation and to systematically decompose it into quality characteristics and sub-characteristics. In accordance to the defined characteristics, we collected data by conducting a user test in order to obtain data on the perceived quality from the point of view of teachers and students, and a correctness test comparing the assessment and grading generated by the CodeMaster tool to manual assessment results. The collected data was analyzed in accordance to the defined analysis questions, using descriptive statistics. The results were interpreted taking also into consideration observations made during the applications.

4. Related Work

Taking into account the importance of (automated) support for the evaluation of practical programming activities in order to improve the teaching of computer science in K-12 education, only very few approaches were found to assess programming activities created with block-based programming languages as detailed in Table 1 (Alves *et al.*, 2017).

Most of the approaches are focused on analyzing Scratch programs. For the assessment of App Inventor or Snap! programs only one approach has been encountered respectively (Maiorana *et al.*, 2015)(Ball, 2017). However, these approaches focus on the assessment of closed-ended well-structured problems with a correct solution known in advance, and are, thus, not applicable directly in problem-based learning context, for ill-structured activities without one single correct solution.

In general, the encountered approaches analyze competences by considering mostly algorithm and programming sub-concepts, such as, algorithms, variables, control and modularity. Some approaches also analyze additional elements, including usability (Denner *et al.*, 2012), code organization and documentation (Denner *et al.*, 2012), aesthetics (Kwon and Sohn, 2016a) (Kwon and Sohn, 2016b) (Denner *et al.*, 2012) and/or creativity (Kwon and Sohn, 2016a)(Kwon and Sohn, 2016b) (Werner *et al.*, 2012).

Most of the approaches use static code analysis approaches counting the frequency of blocks for assessment. Exceptions are approaches aimed at the assessment of problems with known solutions, which are based on tests (Maiorana *et al.*, 2015) (Johnson, 2016) or comparisons with a pre-defined solution (Koh *et al.*, 2014a) (Koh *et al.*, 2014b) (Koh *et al.*, 2010) (Basawapatna *et al.*, 2011). In the context of open-ended ill-structured problems without a single correct solution most approaches use a rubric for assessment, such as Dr. Scratch and Ninja CodeVillage.

Instructional feedback is presented in various forms, such as a total score, grade or in case of Dr. Scratch also through a mascot badge. The approaches also vary in relation to the presentation of only a total result and/or indicating also partial scores in relation to each of the assessed criteria. In addition, Dr. Scratch and Quizly present tips (and tutorials) indicating how to improve the code.

Table 1
Overview on related work

Reference	Approach	Block-based programming language being assessed	Type of educational activity being assessed	Supported by automated software tool for	Type of software support provided
(Kwon and Sohn, 2016a) (Kwon and Sohn, 2016b)	Approach by Kwon & Sohn	Block-based programming languages in general	Open-ended ill-structured problem	--	--
(Franklin *et al.,* 2013) (Boe *et al.,* 2013)	Hairball	Scratch	Closed-ended well-structured problem	Professor	Script-based without graphical user interface
(Moreno and Robles, 2014) (Moreno-León *et al.,* 2016) (Moreno-León and Robles, 2015a) (Moreno-León and Robles, 2015b) (Moreno-León *et al.,* 2017)	Dr. Scratch	Scratch	Open-ended ill-structured problem	Student / Professor/ Institution	Web application with graphical user interface
(Johnson, 2016)	ITCH	Scratch	Closed-ended well-structured problem	Student	Script-based without graphical user interface
(Seiter and Foreman, 2013)	PECT	Scratch	Open-ended ill-structured problem	--	--
(Ota *et al.,* 2016)	Ninja Code Village	Scratch	Open-ended ill-structured problem	Student / Professor	Web application with graphical user interface
(Wolz *et al.,* 2011)	Scrape	Scratch	Open-ended ill-structured problem	Professor	Desktop system with graphical user interface
(Ball and Garcia, 2016) (Ball, 2017)	Autograder	Snap!	Closed-ended well-structured problem	Student / Professor	Web application with graphical user interface
(Maiorana *et al.,* 2015)	Quizly	App Inventor	Closed-ended well-structured problem	Student / Professor/ Administrator	Web application with graphical user interface
(Koh *et al.,* 2014a) (Koh *et al.,* 2014b) (Koh *et al.,* 2010) (Koh *et al.,* 2011) (Basawapatna *et al.,* 2011)	CTP	Agent Sheets	Closed-ended well-structured problem	Professor	not informed
(Werner *et al.,* 2012)	Fairy Assessment	Alice	Closed-ended well-structured problem	--	--
(Denner *et al.,* 2012)	Approach by Denner, Werner & Ortiz	Stagecast Creator	Closed-ended well-structured problem	--	--

Only a few approaches are automated through software tools (Boe *et al.*, 2013) (Moreno-León and Robles, 2015b) (Koh *et al.*, 2014b) (Johnson, 2016) (Ota *et al.*, 2016) (Wolz *et al.*, 2011) (Maiorana *et al.*, 2015). Among these tools, some approaches perform the computation of the commands or compare the student's program with a model solution using static code analysis techniques (Boe *et al.*, 2013) (Moreno-León and Robles, 2015b) (Koh *et al.*, 2014b) (Ota, *et al.*, 2016) (Wolz *et al.*, 2011). Dynamic analysis approaches using tests to validate the student's solution are only adopted in the context of closed-ended well-structured problems (Johnson, 2016) (Maiorana *et al.*, 2015). Half of the encountered approaches are directed towards teacher use, with the specific objective of assessing and grading programming activities. Some tools can be used by both teachers and students, e.g., CTP (Koh *et al.*, 2014a) providing real-time feedback during the programming activity. Only some of the tools provide a web interface facilitating their usage (Moreno and Robles, 2014) (Ota *et al.*, 2016) (Maiorana *et al.*, 2015) (Ball, 2017). Most of the tools are provided as a stand-alone tool, not directly integrated into the programming environment and/or a course management system. Another factor that may hinder their widespread application in practice is their availability in English only, with exception of Dr. Scratch available in several languages.

Thus, we can clearly identify a need for automatic assessment support for other block-based programming languages (such as App Inventor and Snap!) that allow the assessment of open-ended ill-structured problems in problem-based learning contexts.

5. CodeMaster

In order to facilitate the assessment of programming activities in problem-based contexts focusing on learning computational thinking in K-12 education, we developed CodeMaster. CodeMaster is a free web-based system to automatically assess and grade App Inventor and Snap! projects. It focuses on the assessment of educational sound, ill-structured and complex programming activities with no single correct solution, e.g. students developing their own apps to solve transport or healthcare problems in their community or developing their own games with respect to an interdisciplinary topic. It can be used by students to obtain immediate feedback throughout the learning process on their specific project or by teachers in order to assess and grade all programming projects of a whole class being one means in a more comprehensive assessment as suggested by Brennan & Resnick (2012). Furthermore, it can also be used by instructional designers to characterize and create reference/example projects as well as to identify improvement opportunities with respect to instructional units.

We adopt an authentic assessment strategy measuring the students' performance based on the created outcomes of learning activities aiming at programming a software artifact. In order to evaluate whether the outcome produced by students shows that they have learned computational thinking, we use rubrics that indirectly assess the competencies based on measuring indicators of the learning outcome. Following Dr. Scratch (Moreno-León *et al.*, 2016), we measure the complexity of the students'

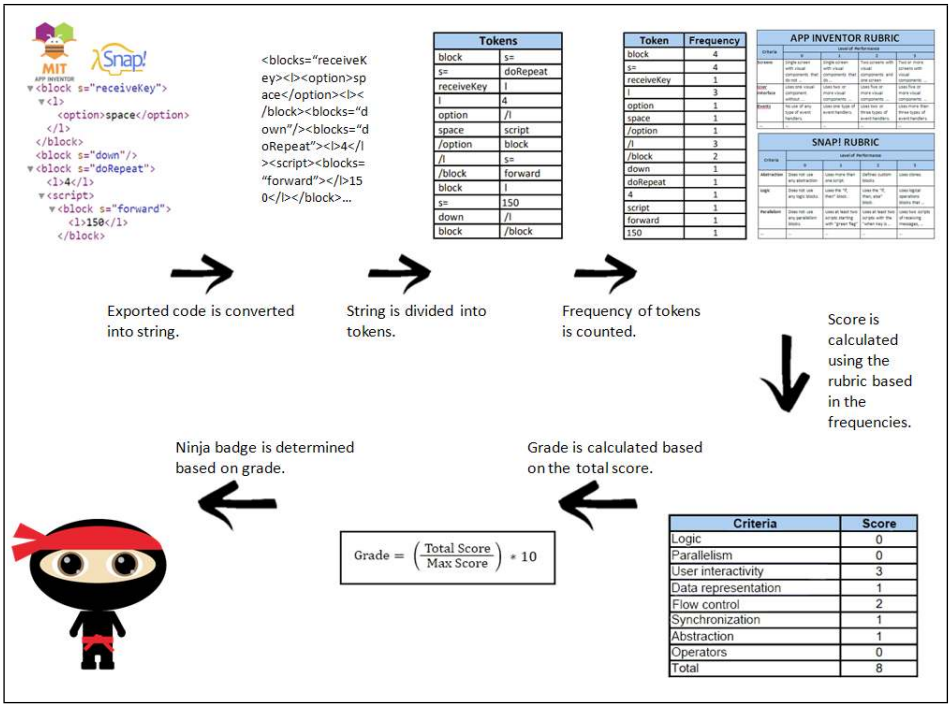


Fig. 4. Overview on the CodeMaster analysis and assessment & grading process

programs with respect to several dimensions of computational thinking, such as abstraction, synchronization, parallelism, algorithmic notions of flow control, user interactivity and data representation based on the CT framework presented by Brennan & Resnick (2012) and the mobile CT rubric (Sherman and Martin, 2015) (Sherman *et al.*, 2014). These dimensions are measured by analyzing the source code of the programs created adopting static code analysis to measure the kind and number of blocks used in the program quantifying CT criteria such as control statement, data, interaction, etc. Then, based on a rubric, the programming projects are assessed and a score and grade is assigned (Fig. 4).

5.1. *CodeMaster Code Analysis*

The CodeMaster tool analyzes the code of programming projects developed with App Inventor 2 (.aia file) and/or Snap! (.xml file). The code analysis is done in three steps:

1. The project code (.aia file or .xml file) is decompressed, read, parsed and converted into a string to be manipulated more easily.
2. A lexical analysis is performed on the resulting string, converting the sequence of characters into a sequence of tokens (strings with an assigned meaning).
3. Then, the tool goes through the token list, counting the frequency of each token, creating a table of tokens and their frequency of use.

5.2. *CodeMaster Project Assessment*

For the assessment, we defined programming language specific rubrics to assess App Inventor (Table 2) and Snap! projects (Table 3). We use analytic rubrics that define criteria for separate individual CT concepts and practices, which allow to calculate a total score based on the individual scores for each criterion. We define the CT criteria to be assessed based on the Computational Thinking Framework presented by Brennan & Resnick (2012) that involves three key dimensions: computational thinking concepts, computational thinking practices, and computational thinking perspectives and has also been adopted by Dr.Scratch (Moreno-Léon *et al.*, 2015a). With respect to the App Inventor rubric, we also take into consideration the Mobile Computational Thinking rubric (Sherman and Martin, 2015) (Sherman *et al.*, 2014), as it extends the CT Framework (Brennan and Resnick, 2012) by adding CT concepts that are present in mobile computing,

Table 2
CodeMaster rubric for assessing SNAP! projects

Criteria	Level of Performance			
	0	1	2	3
Abstraction	Does not use any abstraction blocks.	Uses more than one script.	Defines custom blocks.	Uses clones.
Logic	Does not use any logic blocks.	Uses the “if, then” block.	Uses the “if, then, else” block.	Uses logical operations blocks that combine conditions.
Parallelism	Does not use any parallelism blocks.	Uses at least two scripts starting with “green flag” block.	Uses at least two scripts with the “when key is pressed” block using the same key or two scripts with the “when I’m clicked” block.	Uses two scripts of receiving me-ssages, creating clones or two sensing scripts.
User interactivity	Does not use any user interactivity blocks.	Uses the “green flag” block.	Uses the “key pressed”, “sprite/mouse clicked”, or “ask” block.	Uses “play sound” block.
Data representation	Does not use any data representation blocks.	Uses blocks to modify actor properties such as coordinates, size and appearance.	Uses blocks for operations on variables.	Uses blocks for operations on lists.
Flow control	Does not use any flow control blocks.	Uses a sequence of blocks.	Uses “repeat” or “forever” blocks.	Uses “repeat until” block.
Synchronization	Does not use any synchronization blocks.	Uses the “wait” block.	Uses “say” or “think” blocks with time duration.	Uses “wait until” block.
Operators	No use of any operators blocks.	Uses one type of operator blocks.	Uses two types of operator blocks.	Uses more than two types of operator blocks.

Table 3
CodeMaster rubric for assessing App Inventor projects

Criteria	Level of Performance			
	0	1	2	3
Screens	Single screen with visual components that do not programmatically change state.	Single screen with visual components that do programmatically change state.	Two screens with visual components and one screen with visual components that do programmatically change state.	Two or more screens with visual components and two or more screens with visual components that do programmatically change state.
User Interface	Uses one visual component without arrangement.	Uses two or more visual components without arrangement.	Uses five or more visual components with one type of arrangement.	Uses five or more visual components with two or more types of arrangement.
Naming: Components, Variables, Procedures	Few or no names were changed from their defaults.	10 to 25% of the names were changed from their defaults.	26 to 75% of the names were changed from their defaults.	More than 75% of the names were changed from their defaults.
Events	No use of any type of event handlers.	Uses one type of event handlers.	Uses two types of event handlers.	Uses more than two types of event handlers.
Procedural Abstraction	No use of procedures.	There is exactly one procedure, and it is called.	More than one procedure is used.	There are procedures for code organization and reuse (with more procedure calls than procedures).
Loops	No use of loops.	Uses simple loops (“while”).	Uses “for each” loops with simple variables.	Uses “for each” loops with list items.
Conditional	No use of conditionals.	Uses “if”.	Uses one “if then else”.	Uses more than one “if then else”.
Operators	No use of any operators blocks.	Uses one type of operator blocks.	Uses two types of operator blocks.	Uses more than two types of operator blocks.
Lists	No use of lists.	Uses one single-dimensional list.	Use more than one single-dimensional list.	Uses lists of tuples.
Data persistence	Data are only stored in variables or UI component properties, and do not persist when app is closed.	Data is stored in files (File or FusionTables).	Uses local databases (TinyDB).	Uses web databases (TinyWebDB or Firebase).
Sensors	No use of sensors.	Uses one type of sensor.	Uses two types of sensors.	Uses more than two types of sensors.
Media	No use of media components.	Uses one type of media components.	Uses two types of media components.	Uses more than two types of media components.
Social	No use of social components.	Uses one type of social components.	Uses two types of social components.	Uses more than two types of social components.
Connectivity	No use of connectivity components.	Uses activity starter.	Uses bluetooth connection.	Uses low level web connection.
Drawing and Animation	No use of drawing and animation components.	Uses canvas component.	Uses ball component.	Uses image sprite component.

such as, screen design, location-awareness, and persistent and shared data. We revised the Mobile CT rubric, which has been defined with respect to the technical capabilities of App Inventor Classic, adjusting and adding criteria with respect to new features of App Inventor 2 (such as social and media components).

Different to the CT Framework, we do not include a criterion on sequence, as this is measured based on the simple presence of a sequence of blocks. With respect to the App Inventor rubric, we do not include a criterion on parallelism, as this is not common in app programs also indicated by the fact that is not covered by the Mobile CT rubric. Different to the Mobile CT rubric, we include a criteria related to operators in both rubrics as suggested by the CT framework. Following Dr. Scratch, we also include a criterion on synchronization in the Snap! rubric. Other differences in the App Inventor rubric are basically due to enhancements of the features provided by App Inventor 2, which were not available in App Inventor Classic, when the Mobile CT rubric was defined. Other criteria proposed by the CT framework related to the development process and computational thinking perspectives are not considered as the assessment here is based on the created outcome exclusively.

Each criterion of both CodeMaster rubrics is described along a 4-point ordinal scale, with increasing points representing more sophistication within the concept being measured. For each level of performance of each criterion, we describe observable behaviors as quality definitions. These range from "criterion is not (or minimally) present" to a description of what constitutes advanced usage of the criterion. As a result a score is assigned for each of the criteria, as well as total CT score by the sum of the partial scores. The total CT score ranges from [0; 45] for the assessment of App Inventor projects and from [0; 24] for the assessment of Snap! projects due to the different quantity of criteria assessed:

$$\textit{Total CT score} = \sum \textit{score per criterion}$$

5.3. *CodeMaster Project Grading*

Based on the total CT score a grade is assigned as an indication of the level of performance reflected by the total CT score. CodeMaster assigns grades in two ways, as a numerical grade and a ninja badge.

A numerical grade is assigned in a range from 0.0 to 10.0 by converting the total CT score:

$$\text{Grade} = (\text{score} / \text{maximum score of assessed criteria}) * 10$$

Due to the fact, that not necessarily all programming projects are always expected to include all the defined assessment criteria (especially with respect to mobile apps), we allow teachers to customize the assessment and grading to a specific kind of programming project. Therefore, CodeMaster supports the selection of relevant CT criteria in the assessment of App Inventor programming projects, excluding irrelevant criteria.

CodeMaster also presents the grade in form of a ninja badge on an 10-point ordinal scale of colors of a ninja belt in an engaging way aiming at making the assessment a

Table 4
Definition of Ninja belt color scale

Numerical grade	Ninja belt
0 – 0.9	white
1.0 – 1.9	yellow
2.0 – 2.9	orange
3.0 – 3.9	red
4.0 – 4.9	purple
5.0 – 5.9	blue
6.0 – 6.9	turquoise
7.0 – 7.9	green
8.0 – 8.9	brown
9.0 – 10.0	black

rewarding, challenging and fun part of the learning experience. The color of the ninja badge is based on the numerical grade as indicated in Table 4.

6. Implementation of CodeMaster

Based on the conceptual model the CodeMaster tool has been developed as a web application. The tool automates the assessment and grading of App Inventor and/or Snap! projects. Fig. 5 presents an overview on the use cases implemented by the CodeMaster tool as illustrated in Fig. 6.

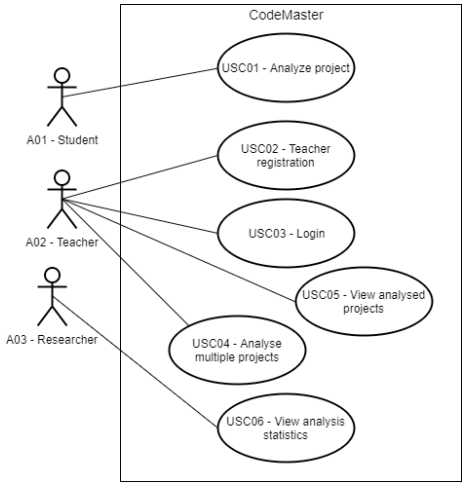
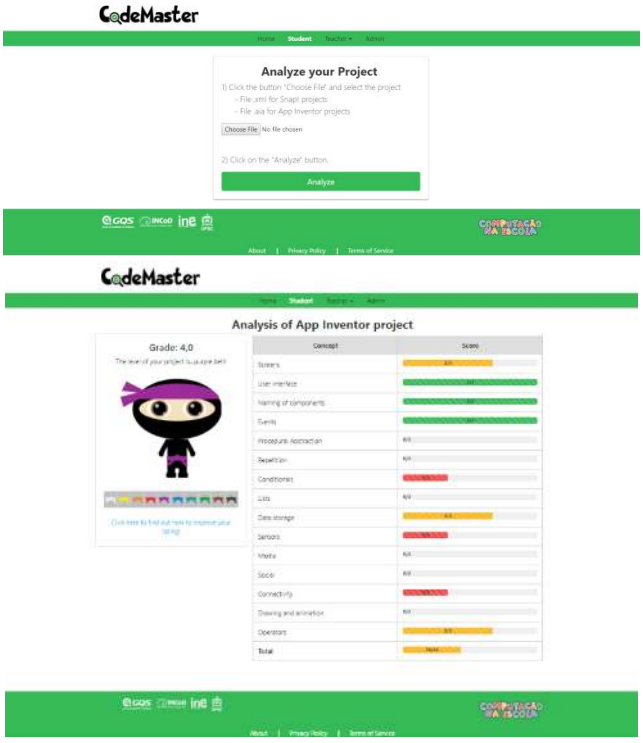
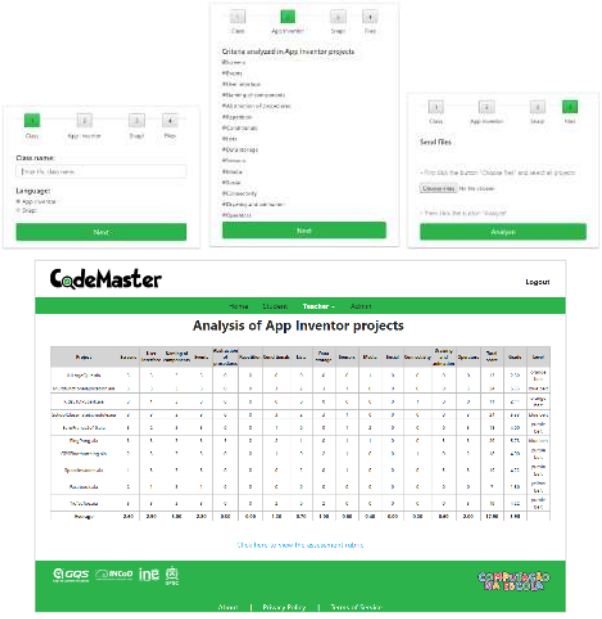


Fig. 5. Use Case diagram[1].

[1] http://www.omg.org/spec/UML/2.5/



(a)



(b)

Fig. 6. (a) Examples of screens of the assessment of individual projects. (b) Examples of screens of the assessment of a whole class.

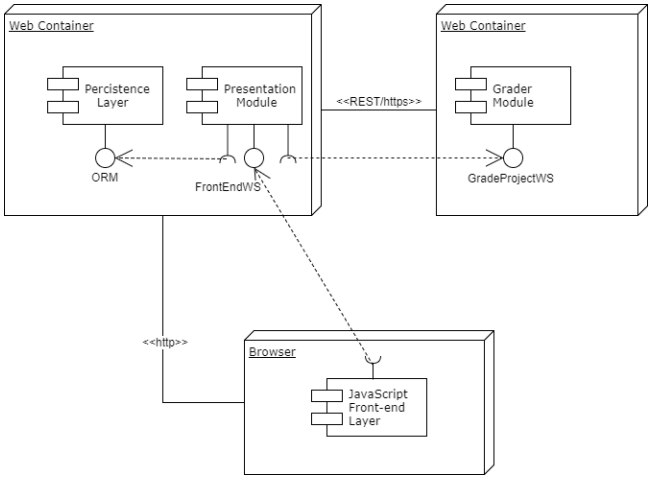


Fig. 7. Overview on CodeMaster's components.

The architectural model of the CodeMaster tool has been defined with the objective of separating presentation and analysis and assessment & grading layers into different modules in order to make the application scalable in the long term and also to allow direct connection of other applications in the future (Fig. 7). The "Analysis & Grader" module is responsible for receiving the project(s), their settings and returning the results of the assessment and grading, through a REST web service. It has been implemented using the Jersey framework (`https://jersey.github.io`) which uses the API JAX-RS (`https://github.com/jax-rs`) abstracting the low-level details of the implementation of the communication between the servers and simplifying the implementation of the REST service. The "Presentation" module is responsible for the user interface, registration of teachers and classes, submission of projects and presentation of results.

The entire backend system was implemented in the Java 8 programming language, running on an Apache Tomcat 8 application server over an Ubuntu 16 operational system, due to our team's competence and the server infrastructure available. The front-end component was implemented in the JavaScript programming language using the Bootstrap library with an additional custom layout. The database used was MySQL 5.7 able to meet the initially estimated demand.

The tool is available online in Brazilian Portuguese and English at:
`http://apps.computacaonaescola.ufsc.br:8080`.

7. Evaluation of CodeMaster

In order to evaluate the quality of the CodeMaster prototype we performed a preliminary evaluation. Our objective is to analyze the quality of the CodeMaster tool in terms of usefulness, functional suitability, performance efficiency and usability from the point

of view of K-12 teachers and students in the context of computing education. Based on ISO/IEC 25010 (2011), ISO/IEC 9241 (1998) TAM (Davis, 1989), and SUS (Brooke, 1996) the quality factors to be evaluated are decomposed (Table 5).

The respective data is collected by conducting a user testing and a correctness test.

Table 5

Overview on the decomposition of the quality characteristics and measurement operationalization

Characteristic	**Sub-characteristic**	**User evaluation**			**Test**
		Questionnaire		**Observation**	
		Teacher questionnaire	**Student questionnaire**		
Usefulness		Do you find the CodeMaster tool useful in computer education in basic education?	Do you find the CodeMaster tool useful for learning programming?		
		Do you think that in its current form (uploading a set of student projects by identifying them by name in the file) the CodeMaster tool is a practical way in your classes?			
Functional suitability	Functional completeness	Do you think there are aspects/criteria for evaluating programming projects in teaching computing in basic education that are not supported by the tool?			
		Do you think that there is any relevant aspects with respect to the process of evaluating programming projects in basic education that are not supported by the CodeMaster tool?			
		Do you think the provided feedback information is sufficient?	Do you think the provided feedback information is sufficient?		
	Functional correctness	Have you noticed any error regarding the functionality of the CodeMaster tool?	Have you noticed any error regarding the functionality of the CodeMaster tool?		Correctness test comparing results from CodeMaster with manual assessment results
			Did you find the assigned grade fair?		
Performance efficiency	Time behavior	Is the performance of the CodeMaster tool satisfactory?	Is the performance of the CodeMaster tool satisfactory?		Performance test

Continued on next page

Table 5 – continued from previous page

Characteristic	Sub-characteristic	User evaluation		Test
		Questionnaire		Observation
		Teacher questionnaire	Student questionnaire	
Usability	Effectiveness			User completed task
	Efficiency			Task completion time
	Satisfaction	I think that I would like to use this system frequently.	I think that I would like to use this system frequently.	
		I found the system unnecessarily complex.	I found the system unnecessarily complex.	
		I thought the system was easy to use.	I thought the system was easy to use.	
		I think that I would need the support of a technical person to be able to use this system.	I think that I would need the support of a technical person to be able to use this system.	
		I found the various functions in this system were well integrated.	I found the various functions in this system were well integrated.	
		I thought there was too much inconsistency in this system.	I thought there was too much inconsistency in this system.	
		I would imagine that most people would learn to use this system very quickly.	I would imagine that most people would learn to use this system very quickly.	
		I found the system very cumbersome to use.	I found the system very cumbersome to use.	
		I felt very confident using the system.	I felt very confident using the system.	
		I needed to learn a lot of things before I could get going with this system.	I needed to learn a lot of things before I could get going with this system.	
	Operability	Did you find the CodeMaster tool easy to use?	Did you find the CodeMaster tool easy to use?	
		Do you think the CodeMaster tool has elements that are ambiguous or difficult to understand?	Do you think the CodeMaster tool has elements that are ambiguous or difficult to understand?	

7.1. *User Testing*

The user testing aims at evaluating the perceived quality from the point of view of teachers and students. During user testing, the users are first given a basic overview on the objective and features of the CodeMaster tool. Then, they perform a predefined task (assessing one or a set of programming projects with the tool). Data is collected through observation as well as a post-test questionnaire. The questionnaire items were

Table 6
Overview on user testing participants

	Elementary/Middle School teachers	Elementary/Middle School students	Total
CodeMaster – App Inventor	3	5	8
CodeMaster – Snap!	4	4	8
Total	7	9	16

derived from the quality characteristics (Table 6). We basically used a nominal scale for the questionnaire items (yes/no), with exception of the items regarding satisfaction. This quality sub-characteristic is measured by adopting the SUS questionnaire (Brooke, 1996) on a 5-point Likert scale. In addition, we also asked them to identify strengths and weaknesses of the tool. The complete material used in the user evaluation is documented in Brazilian Portuguese by Demetrio (2017) and Pelle (2018).

7.1.1. *Execution of User Evaluation*

The evaluation of the tool was carried out by a total of 7 teachers and 9 students in Florianopolis/Brazil (Table 6).

All tests were accompanied by researchers of our initiative *Computação na Escola*/ INCoD/INE/UFSC. The evaluation occurred in September 2017. The data collected is detailed in Appendix A (Demetrio, 2017) (Pelle, 2018).

7.1.2. *Analysis of User Evaluation*

In accordance to the defined quality factors (Table 6), we analyzed the collected data.

Is CodeMaster useful?

All participants considered the CodeMaster tool useful for learning and teaching programming. In general, its contribution to understand the learning of programming has been pointed out as a strength, as it helps the students as well as the teachers to understand if they are learning or not, as well as indicating improvement opportunities. During the test we observed that several students were motivated by a low assessment to immediately continue programming in order to obtain a higher ninja belt. Teachers also emphasized the value of the partial scores with respect to specific criteria providing a detailed feedback. They also recognized that the CodeMaster tool allows to make assessments rapidly and in an organized way. The only issue cited is the way of identification of the individual projects by the names of the students in the teacher module. Teachers suggested that this identification should also include the name of the app and/or project.

Is CodeMaster functional suitable?

The majority of the participants think that all aspects/criteria for evaluating programming projects in teaching computing in basic education are supported by the tool. Only

one teacher suggested that it also should be possible to modify the order of the criteria as well as the projects when presenting the assessment results in the teacher module. Another suggestion with respect to the presentation of the assessment results to the teacher was to visualize the level by a colored image and not only by a label expressing the ninja belt color. One teacher also suggested the consideration of further aspects such as dead code, duplicated code, etc. The way to upload projects was considered positive, but some suggestions were given, such as the possibility of the student sending the project to the teacher directly via the CodeMaster tool.

During the tests we also identified the lack of error messages in case of uploading empty/invalid files and or requesting an assessment without uploading any file. Correcting this issue, the respective error messages have been added.

The majority of the students considered the assigned grade fair. Only one student, trying to improve his grade, indicated that although advancing his program he was not able to improve his grade, which consequently discouraged the student. As a result we increased the number of ninja badge levels from initially 8 to 10 levels in order to facilitate the achievement of a higher level. We also observed during the tests that the primary focus of the student was on the ninja badge, some even not noticing at all that the tool also presented a grade.

Is CodeMaster's performance efficient?

The results of performance tests have shown acceptable average assessment time also confirmed by the participating teachers. However, two students did not agree demonstrating a much lower tolerance on performance delays. One student encountered an efficiency problem uploading a large App Inventor project (including several images) taking about 10 secs to upload probably due to a slow internet connection. In order to improve usability with respect to this issue an indication of the status during upload has been added.

Is CodeMaster usable?

All participants were able to complete successfully the task of assessing one (in the case of students) or a set of projects (in case of the teachers). Applying the System Usability Scale (SUS) (Brooke, 1996) to measure the satisfaction, high scores were given (Bangor *et al.*, 2009) indicating an excellent level of satisfaction (Table 7).

All participants considered the CodeMaster tool easy to use. However one teacher and two students identified elements that were difficult to understand. They indicated

Table 7
Averages of SUS Scores

	Elementary/Middle School teachers	Elementary/Middle School students	Total
CodeMaster – App Inventor	88.13	91.25	89.69
CodeMaster – Snap!	89.17	90.83	90
Total	88.65	91.04	89.84

Table 8
Comments from the participants

Topic	Comments from the teachers	Comments from the students
What did you like with respect to the CodeMaster tool?	Practicality and agility in the evaluation. The design of the tool is excellent, very accessible and easy to use. Within what it proposes to evaluate it is certainly very relevant and practical. It supports the teacher to identify the concepts that the students are able to use and to visualize their progression. Another important aspect is the feature that allows the student to perceive what he learned and what he needs to improve, pointing out ways to overcome difficulties. The items evaluated are useful for us as teachers to think of what we want our students to learn. Collaborate in the integration of curricular content and tool. With the tool it is possible to organize. The functionality and practicality of the tool and the amount of data that can be analyzed.	More possibilities than just Scratch. Easy to use, quick to evaluate and the little doll is cute. I like the little eyes of the doll indicating an error. I was able to do it myself and I liked the doll. The ninja.
Any improvement suggestion with respect to the CodeMaster tool?	Option to choose the concepts that best fit your evaluation objective. Create a gallery on the platform itself. Just make clear what the concepts, scores and grades mean. Option to drag the projects as an e-mail attachment. Especially for those who are not from the area the tool could provide a quick explanation as tool tips on the evaluated items, when presenting the scores. It would be helpful to include a function that allows the sorting of the projects by each of the items, screens, interfaces ... and also total score, grade and ninja belt.	I would like to customize the doll (eye color, ponytail ....) I wanted tips on how to become a black belt and be able to see all my scores/ninja belts (the same way commonly presented by mobile games). On the error page the ninja could get dizzy and fall to the ground. While the tool evaluated the project, you could show the ninja training. Depending on the result the ninja could be happy or sad, e.g., grade 5 and up he becomes happier with each grade, and grade 5 and down he becomes sadder.

difficulty with respect to the understanding of the assessment criteria based on computational thinking concepts and practices, e.g.” I do not know what a loop is”. This indicates that, although, all participants had some knowledge on computing obtained in previous computing workshops, a need for revising the adopted terminology for the instructional feedback is necessary in order to make it more easily understandable. Especially the children liked the ninja being motivated much more to obtain a higher ninja belt level than taking into consideration the numerical grade.

7.2. *Correctness Test*

This test aims at evaluating the correctness of the results generated by the CodeMaster tool comparing the assessment and grading generated by the tool with manual assessment results. For the test we randomly selected 10 apps from the App Inventor Gallery

and 10 Snap! projects from the Snap Galerie (`https://nathalierun.net/snap/Snap.Galerie`). For the manual evaluation of the projects, each selected project was opened in the respective programming environment (App Inventor or Snap!). A performance level was manually assigned with respect to each criteria of the rubric by counting the presence of the blocks and visual components. For the automatic evaluation the projects were uploaded as a set and assessed with the CodeMaster tool. Comparing the results we observed that the same results were obtained in all cases, thus, providing a first indication of the correctness of the CodeMaster tool with respect to the defined rubric.

7.3. *Discussion*

The results of the evaluation provide a first indication that the CodeMaster tool can be a useful, functional, performance-efficient and usable tool to support the assessment of App Inventor and Snap! projects. Principal strengths based on the feedback of the students include the playful way in which the assessment results are presented through the ninja badge. We also observed that getting feedback on their projects motivated them to continue programming trying to improve their assessment. However as observed, the terminology used to provide a detailed feedback per criterion needs to be revised in order to become more understandable by children as well as teachers. Furthermore, in order to guide their improvement more explanations on how the projects can be improved should be presented giving not only an assessment feedback but also guiding the learning process in a personalized way. Although considering the grade fair, some children became frustrated when they were not easily able to improve their grade/ninja belt level continuing programming. Especially with respect to App Inventor projects it can be difficult to achieve higher levels as currently this would require the inclusive of diverse concepts in one app, which not always be necessary depending on the kind of app. Taking into consideration this fact, the teacher module permits to customize the criteria used for the assessment of App Inventor project, yet such a customization is not implemented as part of the student module as the students may not be able to foresee which criteria are relevant, requiring a different solution.

From the viewpoint of the teachers the main strength is the possibility of having a tool that provides support for assessing programming projects of the students in an easy, organized and rapid way. They also emphasised the usefulness of such a tool in the typical school context in Brazil having to attend 30–40 students in each class. The adoption of the tool can also be further facilitated when being integrated into the programming platform itself. Taking into consideration that today computing education is mostly provided by non-computing teachers, they also emphasized the importance of the detailed information per assessment criteria giving not only a general grade, as this allows them to understand more clearly the learning of their students and needs for adjusting teaching.

In general, the assessment criteria are aligned with prominent CT assessment frameworks and similar tools, providing the opportunity to mainly automate the assessment

of computational thinking concept and practices (focusing on programming). Yet, taking into consideration the importance of teaching computing in order to advance 21th century skills (CSTA, 2013) it becomes obvious that a comprehensive performance assessment in computing education should also cover further concepts such as creativity and innovation as well as practices (communication, collaboration etc.) and CT perspectives. In this context, CodeMaster represents only a first step into the direction of automated assessment of programming assignments. Thus, in order to provide a comprehensive feedback, the assessment given by the tool needs to be completed by a manual assessment of the teacher taking into consideration alternative methods as suggested by Brennan and Resnick (2012).

7.4. *Threats to Validity*

The results obtained in this preliminary evaluation need to be interpreted with caution, taking into account potential threats to their validity. One threat may be the research design adopted conducting a series of tests collecting data with respect to utility, functional suitability, and usability through post-test questionnaires and observations. Due to the lack of measurements in a real educational context and a control group, the results are limited to provide only a first indication on the quality of the CodeMaster tool.

The subjects were selected so that their profiles would match the roles of prospective users. However, another threat to validity is related to the sample size, which may compromise the generalizability of the results. Our exploratory study is based on a total of 16 subjects. Such a small sample size hinders any kind of quantitative analysis. However, according to Hakim (1987) small samples can be used to develop and test explanations, particularly in the early stages of the work.

There may also be threats to construct validity. Due to practical limitations, running the study as user tests, the results related to learning effects were obtained from observations of the participants. This type of assessment occurring not within an educational context may not be enough to measure the tool effect. Further evaluation studies within educational contexts are therefore necessary in order to confirm the results. Another possible threat is the definition of the measurement as the quality of software tools is difficult to measure. To counteract this threat, the questionnaires have been developed by systematically decomposing the evaluation goal into questionnaire items adopting the GQM approach (Basili *et al.,* 1994).

8. Conclusion

In this article we present the CodeMaster tool, a web application that analyzes App Inventor or Snap! programs to provide feedback to teachers and students and assigns a CT score and grade to projects. Based on the CT framework and the Mobile CT rubric, we define programming language specific rubrics to conduct performance based assess-

ments of the learning outcome. Adopting a static code analysis approach, we measure the number of programming blocks as performance indicators with respect to CT criteria such as control flow, abstraction, events etc. Based on the rubric a total CT score is calculated that is also converted into a grade. A preliminary evaluation demonstrates that CodeMaster is considered useful, functional, performance-efficient and usable by students and teachers. It can motivate students to continue improving their programs as well as ease the assessment process for teachers. However, we also identified some limitations. An assessment based only on the program project might be limited requiring the usage of other means of assessments approaching CT practices and perspectives. In addition, other competencies, such as creativity or design (being a critical success criterion for apps) are not covered currently. Furthermore, a more detailed personalized feedback on how to improve CT competencies is required in order to better guide and motivate the students. Other issues are related to the implementation of the tool, such as a semantic analysis, coding issues (e.g. dead code, correct use of blocks, etc.) as well as for example the detection of plagiarism. Other improvement opportunities include the inclusion of the approach directly into the programming environment and/or Learning Management Systems, enabling the monitoring of the progress throughout a course. Thus, currently we are working on the improvement of the CodeMaster tool with respect to the identified issues as well as carrying out a series of case studies applying and evaluating the tool in the classroom.

Acknowledgments

We would also like to thank all participants in the evaluation for their valuable feedback.

This work is supported by CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*), an entity of the Brazilian government focused on scientific and technological development.

References

Ala-Mutka, K.M. (2005). A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2), 83–102.

Ala-Mutka, K.M., Järvinen, H.-M. (2004). Assessment process for programming assignments. In: *Proceedings of IEEE Int. Conference on Advanced Learning Technologies.* Joensuu, Finland, 181–185.

Alves, N.d.C., Gresse von Wangenheim, C., Hauck, J.C.R. (2017).Approaches to Assess Computational Thinking Competences Based on Code Analysis in K-12 Education: A Systematic Mapping Study. (in progress)

Ball, M., 2017. Autograding for Snap!. *Hello World*, 3, 26.

Ball, M.A., Garcia, D.D. (2016). Autograding and feedback for Snap!: A visual programming language. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* Memphis, TN, USA, 692–692.

Bangor, A., Kortum, P., Miller, J. (2009). Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4(3), 114–123.

Basawapatna, A.; Koh, K.H.; Repenning, A.; Webb, D.C.; Marshall, K.S. (2011). Recognizing computational thinking patterns. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education.* Dallas, TX, USA, 245–250.

Basili, V.R.; Caldiera, G.; Rombach, D. (1994). Goal question metric approach. *Encyclopedia of Software Engineering.* John Wiley & Sons, 528–532.

Becker, K. (2003). Grading programming assignment using rubrics. In: *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education.* Thessaloniki, Greece, 253–253.

Biggs, J. (2003). *Teaching for Quality Learning at University. 2nd ed*. SRHE/Open University Press, Buckingham.

Black, P. Wiliam, D. (1998). Assessment and classroom learning. *Assessment in Education: Principles, Policy & Practice*, 5(1), 7–74.

Boe, B., Hill, C., Len, M. (2013). Hairball: lint-inspired static analysis of scratch projects. In*: Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. Denver, Colorado, USA, 215–220.

Branch, R.M. (2010). *Instructional Design: The ADDIE Approach*. New York: Springer.

Brennan, K., Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In: *Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada*.

Brooke, J. (1996). SUS: a "quick and dirty" usability scale. In: P.W. Jordan, B. Thomas, B.A. Weerdmeester, A.L. McClelland. *Usability Evaluation in Industry*. London: Taylor and Francis.

Cateté, V., Snider, E., Barnes, T. (2016). Developing a Rubric for a Creative CS Principles Lab. In: *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, Arequipa, Peru*. 290–295.

CSTA (2013). *CSTA K-12 Computer Science Standards: Mapped to Partnership for the 21st Century Essential Skills*. `https://www.ncwit.org/sites/default/files/file_type/2013ce21_cstastandards-mappedtop21centuryskills.pdf`

Davis, F.D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3), 319–340.

DeLuca, C., Klinger, D.A. (2010). Assessment literacy development: identifying gaps in teacher candidates' learning. *Assessment in Education: Principles, Policy & Practice*, 17(4), 419–438.

Demetrio, M.F. (2017). *Development of an App Inventor code analyzer and grader for computing education Project Thesis*. Bachelor of Computer Science Course, Federal University of Santa Catarina, Florianópolis, Brazil.

Denner, J.; Werner, L.; Ortiz, E., (2012). Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education*, 58(1), 240–249.

Douce, C., Livingstone, D., Orwell, J. (2005). Automatic test-based assessment of programming: A review. *Journal on Educational Resourses in Computing*, 5(3), no.4.

Driscoll, A., Wood, S. (2007). *Developing Outcomes-Based Assessment for Learner– Centered Education: a Faculty Introduction*. Stylus Publishing, Sterling, VA, USA.

Eseryel, D., Ifenthaler, D., Xun, G. (2013). Validation study of a method for assessing complex ill structured problem solving by using causal representations. *Educational Technology and Research*, 61, 443–463.

Eugene, K., Stringfellow, C., Halverson, R.(2016). The usefulness of rubrics in computer science. *Journal of Computing Sciences in Colleges*, 31(4), 5–20.

Fee, S.B., Holland-Minkley, A.M. (2010). Teaching Computer Science through Problems, not Solutions. *Computer Science Education*, 2, 129–144.

Fonte, D., da Cruz, D., Gançarski, A.L., Henriques, P.R. (2013). A Flexible Dynamic System for Automatic Grading of Programming Exercises. In: *Proceedings of the 2nd Symposium on Languages, Applications and Technologies*. Porto, Portugal, 129–144.

Forsythe, G.E., Wirth, N. (1965). Automatic grading programs. *Communications of the ACM*, 8(5), 275–278.

Fortus, D., Dershimer, C., Krajcik, J., Marx, R., Mamlok-Naaman, R. (2004). Design-based science and student learning. *Journal of Research in Science Teaching*, 41(10), 1081–1110.

Forsythe, G.E.; Wirth, N. (1965). Automatic grading programs. Communications of the ACM 8(5), 275–278.

Benford, S.D., Burke, E.K., Foxley, E., Higgins, C., (1995). The Ceilidh system for the automatic grading of students on programming courses. In: *Proceedings of the 33rd ACM-SE Annual on Southeast Regional Conference*, Clemson, South Carolina, 176 – 1182.

Franklin, D., Conrad, P., Boe, B., Nilsen, K., Hill, C., Len, M., Dreschler, G., Aldana, G., Almeida-Tanaka, P., Kiefer, B., Laird, C., Lopez, F., Pham, C., Suarez, J., Waite, R. (2013). Assessment of Computer Science Learning in a Scratch-Based Outreach Program. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, Denver, Colorado, USA 371–376.

Funke, J. (2012). Complex problem solving. In: N. M. Seel (Ed.), *The Encyclopedia of the Sciences of Learning*, New York: Springer; Jonassen, 3, 682–685.

Gallagher, S.A. (1997). Problem-based learning: Where did it come from, what does it do, and where is it going? *Journal for the Education of the Gifted*, 20(4), 332–362.

Gijselaers, W.H. (1996). Connecting problembased practices with educational theory. In: L. Wilkerson & W.H. Gijselaers (Eds.), *Bringing Problem-Based Learning to Higher Education: Theory and Practice*. San Francisco: Jossey-Bass, 13–21.

Goodrich, H. (1996). Understanding Rubrics. Educational Leadership, 54(4), 14–18.

Grover, S., Cooper, S., Pea, R. (2014). Assessing computational learning in K-12. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, Uppsala, Sweden, 57–62.

Grover, S., Pea, R. (2013). Computational Thinking in K–1: A review of the state of the field. *Educational Researcher*, 42(1), 38–43.

Grover, S., Pea, R., Cooper, S. (2015). Designing for deeper learning in a blended computer science course for middle school students. *Journal Computer Science Education*, 25(2), 199–237.

Guindon, R. (1988). Software design tasks as ill-structured problems, software design as an opportunistic process. Microelectronics and Computer Technology Corporation, Austin, TX, USA.

Hakim, C. (1987). Research Design: *Strategies and Choices in the Design of Social Research. Contemporary Social Research: 13, ed.* M. Bulmer, London: Routledge.

Hart, D. (1994). *Authentic Assessment: A Handbook for Educators*. Addison-Wesley Pub. Co, Menlo Park, CA.

Harvey, B., Garcia, D., Paley, J., Segars, L. (2012). Snap!:(build your own blocks). In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, Raleigh, NC, USA.*

Harvey, B., Mönig, J. (2017). "Snap! Reference Manual. `https://people.eecs.berkeley.edu/~bh/byob/SnapManual.pdf`

Hattie, J., Timperley, H. (2007). The power of feedback. *Review of Educational Research*, 77(1), 81–112.

Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli, Finland, 86–93.

ISO/IEC 25010. (2011). Systems and software engineering –Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

ISO 9241-11. (1998) Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability.

Johnson, D. E., (2016). ITCH: Individual Testing of Computer Homework for Scratch Assignments. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education,* Memphis, Tennessee, USA, 223–227.

Kafai, Y., Burke, Q. (2013). Computer programming goes back to school. *Phi Deltan Kappan*, 95(1), 61–65.

Kelleher, C., Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2), 83–137.

Keuning , H., Jeuring, J., Heeren, B. (2016). Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In: *Proc. of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, Arequipa, Peru, 41–46.

Kindborg, M., Scholz, R. (2006). MagicWords – A programmable learning toy. In: *Proceedings of the 2006 Conference on Interaction design and children*, Tampere, Finland, 165–166.

Kitchenham, B.A, Budgen, D., Brereton, O.P., (2011). Using mapping studies as the basis for further research – A participant-observer case study. *Information and Software Technology*, 53(6), 638–651.

Koyya, P., Lee, Y., Yang, J. (2013). Feedback for programming assignments using software-metrics and reference code. *ISRN Software Engineering*, article id 805963.

Koh, K.H., Basawapatna, A.,Bennett, V., *et al.*, (2010). Towards the automatic recognition of computational thinking. In: *Proceedings of the IEEE International Symposium on Visual Languages and Human-Centric Computing*, Madrid, Spain, 59–66.

Koh, K.H., Bennett, V., Repenning, A., (2011). Computing indicators of creativity. In: *Proceedings of the 8th ACM Conference on Creativity and Cognition Atlanta*, Georgia, USA, 357–358.

Koh, K.H., Basawapatna, A., Nickerson, H., Repenning, A. (2014a). Real time assessment of computational thinking. In: *Proceedings of the IEEE Symposium onVisual Languages and Human-Centric Computing*, Melbourne, Australia, 49–52.

Koh, K.H., Nickerson, H., Basawapatna, A., Repenning, A., (2014b). Early validation of computational thinking pattern analysis. In: *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, Uppsala, Sweden, 213–218.

Kwon, K.Y., Sohn, W.-S. (2016). A framework for measurement of block-based programming language. *Asia-Pacific Proceedings of Applied Science and Engineering for Better Human Life*, 10, 125–128.

Kwon, K.Y. and Sohn, W-S. (2016). A method for measuring of block-based programming code quality. *International Journal of Software Engineering and Its Applications*, 10(9), 205–216.

Larman C., Basili, V. (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer*, 36(6), 47–56.

Lye, S. Y., Koh, J.H.L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12?. *Computers in Human Behavior*, 41(C), 51–61.

Maiorana, F., Giordano, D., Morelli, R., (2015). Quizly: A live coding assessment platform for App Inventor. In: *Proceedings of IEEE Blocks and Beyond Workshop*,Atlanta, GA, USA, 25–30.

McCauley, R. (2003). Rubrics as assessment guides. *Newsletter ACM SIGCSE Bulletin*, 35(4),17–18.

Merrill, D.C., Reiser, B.J., Ranney, M., Trafton, J.G. (1992). Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *Journal of the Learning Sciences*, 2(3), 277–305.

Monroy-Hernández, A., Resnick, M. (2008). Empowering kids to create and share programmable media Interactions. 15(2), 50–53.

Moreno, J., Robles, G., (2014). Automatic detection of bad programming habits in scratch: A preliminary study. In: *Proceeding of Frontiers in Education Conference*, Madrid, Spain, 1–4.

Moreno-León, J., Robles, G. (2015a). Analyze your Scratch projects with Dr. Scratch and assess your Computational Thinking skills. In: *Proceedings of the Scratch Conference*, Amsterdam, Netherlands, 1–7.

Moreno-León, J., Robles, G. (2015b). Dr. Scratch: a Web Tool to Automatically Evaluate Scratch Projects. In: *Proceedings of the 10th Workshop in Primary and Secondary Computing Education*, London, UK, 132–133.

Moreno-Léon, J., Robles, G., Román-González, M. (2015). Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. RED-Revista de Educación a Distancia, 4615.

Moreno-León, J., Robles, G., Román-González, M. (2016). Comparing computational thinking development assessment scores with software complexity metrics. In: *Proceedings of IEEE Global Engineering Education Conference*, Abu Dhabi, UAE, 1040– 1045

Moreno-León, J., Román-González, M., Harteveld, C., Robles, G.(2017). On the Automatic Assessment of Computational Thinking Skills. In: *Proc. of the Conference on Human Factors in Computing Systems Denver*, Colorado, USA, 2788–2795.

Mustafaraj, E., Turba, F., Svanberg, M. (2017). Identifying Original Projects in App Inventor. MIT. App Inventor Classic. `http://appinventor.mit.edu/explore/classic.html`

Ota, G., Morimoto, Y., Kato, H. (2016). Ninja code village for scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. In: *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*, Cambridge, UK.

Pelle, R. (2018). Development of a Snap! code analyzer and grader for computing education Project Thesis, Bachelor of Computer Science Course, Federal University of Santa Catarina, Florianópolis, Brazil. (in progress)

Petersen, K., Vakkalanka, S., Kuzniarz, L. (2015). Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64(C), 1–18.

Popham, W.J. (2009). Assessment literacy for teachers: Faddish or fundamental? *Theory into Practice*, 48(1), 4–11.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60–67.

Romli, R., Sulaiman, S., Zamli, K.Z. (2010). Automatic programming assessment and test data generation – a review on its approaches. In: *Proceedings of International Symposium in Information Technology*, Kuala Lumpur, Malaysia, 1186–1192.

Sadler, D.R. (1989). Formative assessment and the design of instructional systems, *Instructional Science*, 18(2), 119–144.

Seiter, L., Foreman, B.(2013). Modeling the Learning Progressions of Computational Thinking of Primary Grade Students. *Proceedings of the 9th Annual Int. ACM Conference on International Computing Education Research*, San Diego, California, USA, 59–66.

Shelton, J.B., Smith, R.F. (1998). Problem-based learning in analytical science undergraduate teaching. *Research in Science and Technological Education*, 16(1), 19–29.

Sherman, M., Martin, F. (2015). The assessment of mobile computational thinking. *Journal of Computing Sciences in Colleges*, 30(6), 53–59.

Sherman, M., Martin, F., Baldwin, L., DeFilippo, J. (2014). *App Inventor Project Rubric – Computational Thinking through Mobile Computing*. `https://nsfmobilect.files.wordpress.com/2014/09/mobile-ct-rubric-for-app-inventor-2014-09-01.pdf`

Shute. V.J. (2008). Focus on formative feedback. *Review of Educational Research*, 78(1),153–189.

Simon, H.A. (1983). The structure of ill-structured problems. *Artificial Intelligence*, 4, 181–201.

Smith, L., Cordova, J. (2005). Weighted primary trait analysis for computer program evaluation. *Journal of Computing Sciences in Colleges*, 20(6), 14–19.

Srikant, S., Aggarwal, V. (2013). Automatic Grading of Computer Programs: A Machine Learning Approach. *Proceeding of 12th International Conference on Machine Learning Applications*, Miami, FL, USA.

Srikant, S., Aggarwal , V. (2014). A System to Grade Computer Programming Skills using Machine Learning, *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA,1887–1896.

Torrance, H. (1995). *Evaluating Authentic Assessment: Problems and Possibilities in New Approaches to Assessment*. Buckingham: Open University Press.

Turbak, F., Mustafaraj, F., Svanberg, M., Dawson, M. (2017). Identifying and analyzing original projects in an open-ended blocks programming environment. In: *Proc. of the 23rd Int. Conference on Visual Languages and Sentient Systems.* Pittsburgh, PA, USA.

Vujosevic-Janicic, M., Nikolic, M., Tosic, D., Kuncak, V. (2013). On software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology*, 55(6), 1004–1016.

Ward, J.D. & Lee, C.L. (2002). A review of problem-based learning. *Journal of Family and Consumer Sciences Education*, 20(1), 16–26.

Weintrop, D. Wilensky, U., (2015). To Block or not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. In: *Proceedings of the 14th International Conference on Interaction Design and Children*, Boston, Massachusetts, USA, 199–208.

Werner, L., Denner, J., Campe, S., Kawamoto, D.C., (2012). The Fairy performance assessment: Measuring computational thinking in middle school. In: *Proceedings of ACM Technical Symposium on Computer Science Education*, Raleigh, North Carolina, USA, 215–220.

Whittaker, C.R., Salend, S.J., Duhaney, D. (2001). Creating instructional rubrics for inclusive classrooms. *Teaching Exceptional Children*, 34(2), 8–13.

Wiggins, G.P. (1993). The Jossey-Bass education series. Assessing student performance: Exploring the purpose and limits of testing. San Francisco: Jossey-Bass.

Wilcox, C. (2016). Testing Strategies for the Automated Grading of Student Programs. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, Memphis, Tennessee, USA, 437–442.

Wing, J. (2006). Computational Thinking. *Communications of the ACM*, 49(3), 33–36.

Wolber, D., Abelson, H., Friedman, M. (2014). Democratizing computing with App Inventor. *Get Mobile*, 18(4), 53–58.

Wolz, U., Hallberg, C., Taylor, B. (2011). Scrape: A tool for visualizing the code of scratch programs. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, Dallas, TX, USA.

Yadav, A., Burkhart, D., Moix, D., Snow, E., Bandaru, P., Clayborn, L. (2015). Sowing the Seeds: A Landscape Study on Assessment in Secondary Computer Science Education. In: *Proceedings of CSTA Annual Conference, Grapevine*, TX, USA.

Zen, K., Iskandar, D.N.F.A., Linang, O. (2011). Using Latent Semantic Analysis for automated grading programming assignments. In: *Proc. of the 2011 International Conference on Semantic Technology and Information Retrieval*. Kuala Lumpur, Malaysia, 82–88.

C.G. von Wangenheim, is a professor at the Department of Informatics and Statistics (INE) of the Federal University of Santa Catarina (UFSC), Florianópolis, Brazil, where she coordinates the Software Quality Group (GQS) focusing on scientific research, development and transfer of software engineering models, methods and tools and software engineering education in order to support the improvement of software quality and productivity. She also coordinates the initiative Computing at Schools, which aims at bringing computing education to schools in Brazil. She received the Dipl.-Inform. and Dr. rer. nat. degrees in Computer Science from the Technical University of Kaiserslautern (Germany), and the Dr. Eng. degree in Production Engineering from the Federal University of Santa Catarina. She is also PMP – Project Management Professional and MPS. BR Assessor and Implementor.

J.C.R. Hauck holds a PhD in Knowledge Engineering and a Master's Degree in Computer Science from the Federal University of Santa Catarina (UFSC) and a degree in Computer Science from the University of Vale do Itajaí (UNIVALI). He held several specialization courses in Software Engineering at Unisul, Univali, Uniplac, Uniasselvi, Sociesc and Uniarp. He was a visiting researcher at the Regulated Software Research Center – Dundalk Institute of Technology – Ireland. He is currently a Professor in the Department of Informatics and Statistics at the Federal University of Santa Catarina.

M.F. Demetrio is an undergraduate student of the Computer Science course at the Department of Informatics and Statistics (INE) of the Federal University of Santa Catarina (UFSC) and a research student at the initiative Computing at Schools/INCoD/INE/UFSC.

R. Pelle is an undergraduate student of the Computer Science course at the Department of Informatics and Statistics (INE) of the Federal University of Santa Catarina (UFSC) and a research student at the initiative Computing at Schools/INCoD/INE/UFSC.

N. da Cruz Alves is an master student of the Graduate Program in Computer Science (PPGCC) at the Federal University of Santa Catarina (UFSC) and a research student at the initiative Computing at Schools/INCoD/INE/UFSC.

H. Barbosa is an undergraduate student of the Design course at Department of Graphic Expression of the Federal University of Santa Catarina (UFSC) and a scholarship student at the initiative Computing at Schools/INCoD/INE/UFSC.

L.F. Azevedo is an undergraduate student of the Design course at Department of Graphic Expression of the Federal University of Santa Catarina (UFSC) and a scholarship student at the initiative Computing at Schools/INCoD/INE/UFSC.

Appendix A. Responses collected during user testing via questionnaire

Characteristic	Sub-characteristic	Teacher questionnaire	Yes	No	Student questionnaire	Yes	No
Usefulness		Do you find the CodeMaster tool useful in computer education in basic education?	7	0	Do you find the CodeMaster tool useful for learning programming?	7	0
		Do you think that in its current form (uploading a set of student projects by identifying them by name in the file) the CodeMaster tool is a practical way in your classes?	5	2			
Functional suitability	Functional completeness	Do you think there are as-pects/ criteria for evaluating programming projects in teaching computing in basic education that are not supported by the tool?	2	5			
		Do you think that there is any relevant aspects with respect to the process of evaluating programming projects in basic education that are not supported by the CodeMaster tool?	2	5			
		Do you think the provided feedback information is sufficient?	6	1	Do you think the provided feedback information is sufficient?	7	0
	Functional correctness	Have you noticed any error regarding the functionality of the CodeMaster tool?	1	6	Have you noticed any error regarding the functionality of the CodeMaster tool?	1	6
					Did you find the assigned grade fair?	6	1
Performance efficiency	Time behavior	Is the performance of the CodeMaster tool satisfactory?	7	0	Is the performance of the CodeMaster tool satisfactory?	5	2
Usability	Effectiveness	Task completed	7	0		7	0
	Satisfaction	Average SUS score	Table 7		Average SUS score	Table 7	
	Operability	Did you find the CodeMaster tool easy to use?	7	0	Did you find the CodeMaster tool easy to use?	7	0
		Do you think the CodeMaster tool has elements that are ambiguous or difficult to understand?	1	6	Do you think the CodeMaster tool has elements that are ambiguous or difficult to understand?	2	5