Rubric for the qualitative assessment of student-designed Snap! Projects

Nicole Marmé[1], Jens-Peter Knemeyer[1], Alexandra Švedkijs[1]

[1]University of Education Heidelberg

DOI: 10.21585/ijcses.v7i3.226

Abstract

An objective evaluation and assessment of individual student-designed projects are challenging. Appropriate tools are currently lagging and have to be developed. Block-based programming languages, such as Snap! are often used for teaching programming basics and the subsequent development of student-designed programming projects. The current research qualitatively developed a rating rubric for Snap! projects to investigate how novices' programming skills can be evaluated and assessed in a criterion-guided manner. For this purpose, an evaluation was conducted on a baseline dataset of 36 student projects created over three school years after a programming course for novices. Based on this database we designed an assessment rubric. A team of experts reviewed and evaluated the assessment rubric. Following expert evaluation, the rubric was improved and expanded. Finally, prospective teachers conducted a comparative evaluation of a test data set consisting of ten Snap! projects of varying complexity, with and without the resulting rubric. The results show that the rating rubric significantly improves the comparability of assessments. In addition, a clear differentiation of the projects by level is achieved for the test data set. Furthermore, the assessment rubric enables a more precise achieved result evaluation in particular rubric category.

Keywords: Computer science, rubric, qualitative assessment, learning outcomes, teaching materials, coding, programming languages, Snap!

1. Introduction

Global challenges and technological progress have brought about a heightened emphasis on information technology skills over the last two decades. The demand for e-learning offers is constantly growing, especially for IT skills (mmb Institut, 2021). Digitization at all levels and global crises, such as the Covid-19 pandemic,

are intensifying discussions across Europe about which skills and abilities will be needed in the future to be able to participate in social life (European Commission. Directorate General for Communication., 2020). Regarding digital competences in particular, competence requirements and necessary action steps for the next decades are being formulated nationally and internationally at various political levels (European Commission. Directorate General for Education, Youth, Sport and Culture., 2023). The Council of the European Union highlights digital literacy as one of the eight key competences for lifelong learning in the 21st century (Publications Office of the European Union, 2019). The current framework on European Union digital citizenship competence DigComp 2.2 lists programming competence as one of the key competences (European Commission. Joint Research Centre., 2022). The current approach to facing the challenges in Germany, for example, is to expand computer science lessons across all grades from fifth grade onwards. For the required strengthening of programming skills, the current educational plan for computer science recommends block-based programming for the acquisition of basic knowledge and skills in programming, especially for beginners (Ministerium für Kultus, Jugend und Sport Baden-Württemberg, 2016a, 2016b). The use of block-based programming languages often goes hand in hand with the development of individual creative projects (Krugel & Ruf, 2020; Resnick, Silverman, et al., 2009; Resnick, 2014). To successfully implement block-based programming languages in the classroom, a systematic approach is needed to evaluate such creative student projects. There are already some approaches to evaluating block-based programmes as will be discussed in section 2.2. However, most approaches deal with automated evaluation of the generated code. This involves solving pre-designed test tasks and evaluating them automatically. Such systems do not allow for the evaluation of individual projects on open topics. This paper therefore investigates whether a competency grid can be used to evaluate open-ended Snap! projects and how such a grid must be structured to ensure valid and consistent evaluation. To address this question, a multi-phase research design was applied, including the development of the rubric, expert validation, and empirical testing with student projects. The results demonstrate that the rubric improves the comparability of evaluations and provides a practical, criterion-based tool for assessing creative, block-based programming projects.

2. Background

2.1 Block-based programming for novices

Block-based programming languages are visual programming languages that use blocks to represent code, rather than traditional text-based code. This allows users to create programs by dragging and dropping these blocks together, without having to write lines of code. A program code is put together like a puzzle by assembling the already available instruction blocks. These environments operationalize Papert's constructionist principles by providing concrete, manipulable elements that support self-directed creation, experimentation, and reflection (Papert, 1993). Learners actively construct knowledge, explore multiple solution paths, and iteratively refine their projects, fostering discovery-based learning and reducing the abstraction barriers typical of traditional coding (Brennan & Resnick, 2012; Resnick et al., 2009). Platforms such as Snap! enable

students to design interactive projects-games, stories, or animations - promoting cognitive engagement, problem-solving, and creativity. The visual, block-based interface simplifies syntax, while project sharing, remixing, and collaborative exploration enhance social learning and knowledge co-construction, key aspects of constructivist and constructionist pedagogy (Papavlasopoulou et al., 2019). Compared to common programming languages that use textual syntax, block-based languages allow easier interaction with the programming environment and learners can focus more on programming logic instead of dealing with syntactical errors (Balouktsis, 2016). Block-based languages provide a low barrier to entry and a flexible, expressive environment. This allows learners to focus on creative and meaningful projects, fostering computational thinking, systematic reasoning and digital literacy (Resnick, Maloney, et al., 2009).

Block-based programming languages are characterised by their ability to eliminate syntax errors, reduce cognitive load and shift the focus from memory recall to visual recognition through structured, visual program construction. They are particularly valuable in lowering the entry barrier for novices and enabling intuitive, interactive learning that fosters engagement and a deeper understanding of core programming concepts (Bau et al., 2017). In particular, beginners are able to concentrate more on understanding programming concepts rather than memorising text syntax due to the reduction in cognitive load (Weintrop & Wilensky, 2018)

Block-based programming languages, such as Snap! show significant advantages for introducing programming to novices. These languages are considered "easier" than text-based programming languages (Weintrop & Wilensky, 2015) and enable an introduction to programming for learners without any prior knowledge (Maloney et al., 2010). For example, the use of block-based programming languages can provide a better understanding of basic programming concepts, like loops (Mladenović et al., 2020). In addition, block-based programming languages offer a more visual interface that can make programming concepts more accessible. Features such as execution visibility, language extensibility and liveness in block-based languages create a positive attitude towards learning and using them (Perera et al., 2021). The use of block-based languages also increases student motivation in introductory programming courses by promoting positive emotions about performance, which in turn improves learning performance and engagement (Tsai, 2019; Wen et al., 2023). With block-based programming languages, learners grasp the task more quickly and achieve significantly more learning goals in the same amount of time compared to those using text-based languages (Price & Barnes, 2015). Interest in further programming activities is also rated higher after a learning sequence with a block-based programming language (Weintrop & Wilensky, 2017). The integration of block-based programming activities significantly improves pupils' computational thinking skills and their self-efficacy in problem solving. Such activities actively engage learners, promote their independence and strengthen their confidence in applying programming concepts (Koray & Bilgin, 2023).

Snap! is a further development of the Scratch programming environment, already established in many schools. Snap! offers some advantages and additional functions compared to Scratch; for example, Snap! enables comprehensive prototype-based programming by creating objects (Modrow, 2018). In addition, new blocks can be created as subroutines with control structures, also called the Build Your Own Block principle.

The programming toolbox for object-oriented programming is comprehensive, so that Snap!, in contrast to Scratch, is a "fully developed programming language" (Modrow, 2018) and is thus in principle also suitable for advanced computer science teaching. This is also reflected in the fact that Snap! is now sometimes offered as an introductory programming language for first semesters of computer science (Garcia et al., 2012). In summary, learning programming using block-based programming languages such as Snap! offers an accessible and visual approach to learning basic concepts, enabling students to develop essential programming skills while fostering their creativity, problem-solving abilities, and logical reasoning. Block-based programming languages are moreover based on the vision of enabling programming beginners to implement learning-by-doing or learning-by-making, where they are free to experiment with their own ideas, such as creating, sharing, playing, and learning with computers (Harel et al., 1993). Therefore, to promote programming skills for beginners in a school context, the use of block-based programming languages can be beneficial, especially for creation of student-designed projects.

2.2 Assessment of block-based programmed student projects

When working in the context of student-designed projects, it is crucial to establish suitable evaluation concepts that offer clear and transparent assessment measures for both teachers and students. By doing so, educators can review the quality of learning materials and provide valuable feedback to support student learning and growth. Assessment of student performance and feedback is an essential part of the learning process (Hattie, 2009). Nevertheless, the assessment process is one of the most complex activities in a teacher's job (Jürgens & Lissmann, 2015). Effective feedback should focus on the task and process, provide clear guidance on how to improve, link specifically to goals and performance (Shute, 2008). Additionally, research suggests that feedback should be specific and focused on the most important aspects of student work (Wiliam, 2011). The challenge of assessing student-designed projects lies in their open-ended nature, as they are characterized by diverse approaches, ideas, and implementations, making direct comparisons difficult.

To address this challenge, various concepts and tools for assessing block-based programming projects have been proposed. In most assessment concepts, however, there is a lack of consensus regarding the concrete establishment and weighting of assessment criteria (Da Cruz Alves et al., 2019). This is probably because there is currently no standardized competence framework derived from an empirically validated model (Gesellschaft für Informatik, 2016). Moreover, most existing systems were not designed for the evaluation of authentic, open-ended projects, but rather for standardized, task-based learning contexts**.** Most authors concerned with assessment, either through the development of tools or the investigation of evaluation processes, regard their approaches as supplementary to teaching and as a means of supporting learning (Boe et al., 2013; Denner et al., 2012; Funke & Geldreich, 2017; Koh et al., 2014; Moreno-León et al., 2017; Seiter & Foreman, 2013; Werner et al., 2012; Zhang & Biswas, 2019).

Table 1 provides an overview of prominent approaches and tools for assessing block-based programming projects, highlighting their aims, methods, strengths, and limitations.

Study (Author, Year)	Aim / Context	Assessment Method	Strengths	Limitations
Boe et al., 2013 – Hairball	Evaluate Scratch projects to identify problematic or missing constructs	Static analysis with customizable plugins (e.g., initialization, synchronization, loops)	Objective, scalable detection of code patterns; high accuracy (≈99%)	Limited to predefined patterns; cannot assess creativity or design; manual review still needed
Denner et al., 2012	Middle school girls' game projects (Stagecast Creator)	Research study analysing 108 games using coding categories (complexity, usability, documentation)	Authentic insight into conceptual understanding; large dataset	Not a standardized tool; rule-based, not block-based; limited transferability
Koh et al., 2014 – REACT	Middle school STEM / Scalable Game Design classes	Real-time formative assessment of computational thinking patterns	Timely feedback for teachers; identifies misconceptions during coding	Limited to predefined CT patterns; misses qualitative and creative aspects
Werner et al., 2012 – Fairy Performance Assessment	Game programming elective using Alice	Performance-based tasks measuring CT (abstraction, modelling, problem-solving)	Authentic, multi-dimensional CT measurement; supports collaboration studies	Specific to Alice; high implementation effort; limited creativity assessment
Ball & Garcia, 2016 – Autograder λ	University Snap! courses	Automated grading and feedback integrated into Snap!	Scalable grading; immediate feedback; simple setup	Limited to closed-ended tasks; no assessment of creativity or design
Wang et al., 2021 – SnapCheck	Snap! courses with interactive projects	Automated testing using predefined templates and simulated user actions	High accuracy (≈98%); scalable; integrated into Snap!	Only for testable behaviours; setup time-intensive; cannot assess open-ended creativity
Moreno-León et al., 2017 – Dr. Scratch	Scratch programming contest projects	Automated static analysis compared to human expert ratings	Strong correlation with experts; consistent and scalable	Ignores creativity and design; focused on technical aspects only

As the table (Table 1) shows, automated tools such as Hairball (Boe et al., 2013), Dr. Scratch (Moreno-León et al., 2017), or SnapCheck (Wang et al., 2021) offer highly scalable solutions and produce consistent results but are primarily limited to predefined technical patterns and cannot capture creativity or the quality of open-ended designs. In addition, some systems face technical barriers such as installation issues and a constant need for updates to remain functional, which affects their acceptance among teachers (e.g., Ball & Garcia, 2016). Even when functioning well, these systems often provide only structural feedback about the code and lack the ability to evaluate whether a problem was solved in a meaningful way (Moreno-León et al., 2017; Wang et al., 2021). These limitations explain why most authors explicitly recommend using automated systems as a complement to traditional, teacher-driven assessments rather than as a replacement. For example, Hairball and Dr. Scratch are powerful tools for detecting certain constructs, but they do not

assess design aspects, while SnapCheck provides highly accurate testing of interactive behaviours yet requires significant preparation of templates and is unsuitable for authentic, free-topic projects. Thus, there is still a clear need for research and development to create evaluation instruments that can provide rich, individualized feedback for authentic student work.

One effective approach to evaluating student-designed projects is using rubrics. Andrade H. defines a rubric as a one– or two–page document that describes varying levels of quality, from excellent to poor, for a specific assignment (Andrade, 2000). Rubrics provide a clear and consistent framework for evaluating authentic student-designed projects. By making expectations explicit and providing qualitative, criterion-based feedback, rubrics help students understand how to improve their work and promote deeper learning (Wolf & Stevens, 2007). The rubric presented in this study was developed specifically for Snap! projects and aims to qualitatively capture and objectively assess the outcomes of open-ended, autonomous student projects. It was developed as part of the evaluation of an interdisciplinary self-learning course, "Smart City" (Svedkijs et al., 2022) for learning the basics of programming with Snap! to be able to qualitatively record and objectively assess the student projects created.

3. Method

3.1 Development procedure

We opted for a qualitative and exploratory approach to developing the assessment rubric because the research question is open and the aim is to generate a practical, field-tested assessment instrument (Döring & Bortz, 2016; Gummels, 2020).

3.2 Teaching sequence

To this purpose, 183 students (the majority with no prior knowledge) in grades 9-11 were taught the basics of programming with the block-based language Snap! in an approx. 20-hour teaching sequence in the school years 2018/19, 2019/20 and 2020/21. No one had any previous knowledge of block-based programming. Following the lesson sequence, the pupils created their own projects in small groups on a free topic. Forty pupil projects resulted from this and after data cleansing, 36 projects were available. The rubric was developed using anonymized student project data, collected with informed consent and without any personal identifiers, complemented by published projects from the Snap! platform.

3.3 Analysis and Drafting

Available projects could be used as a baseline data set for the development of the rubric (Fig.1). The development of the rubric involved a comprehensive process, starting with the analysis of baseline data from student projects and expert evaluation.

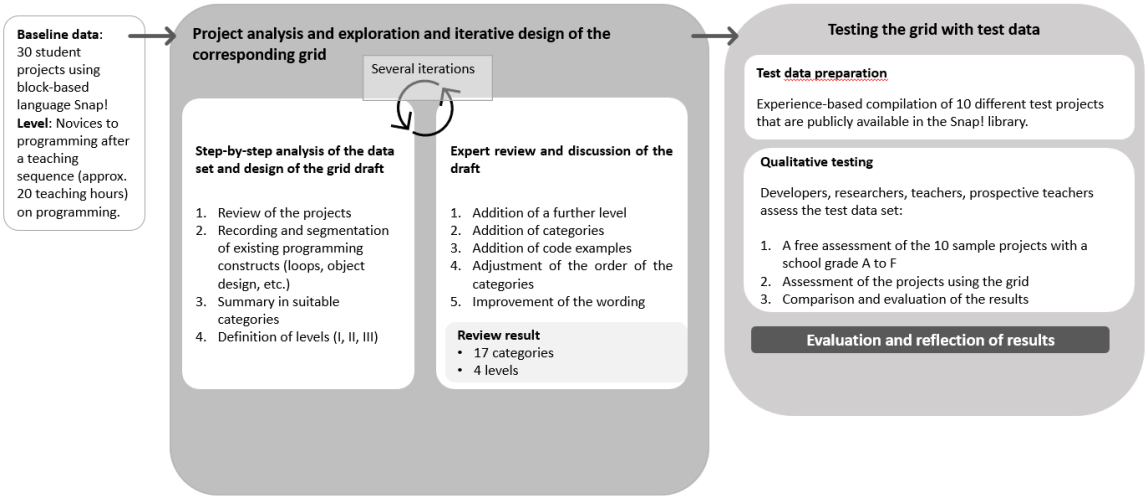


Figure 1:Development process of the assessment rubric

To begin, a thorough analysis of the given dataset was conducted, examining each project's structure and content to gain a deep understanding of its programming constructs, such as loops and object designs. This allowed for systematization and categorization of used programming constructs. The resulting summaries enabled definition of three different levels (I, II, III) within the dataset. Based on these findings, a draft of the rubric was created with twelve thematic categories and three levels.

3.4 Expert Review and Iterative Exchange

We reviewed the initial rubric version together with four educational experts (2 female, 2 male) in the field of programming for qualitative assessment. We defined experts as individuals with several years of experience using block-based languages, particularly Snap!, in teaching contexts or those who had published academically on block-based programming languages. Experts' review led to refinement through an iterative exchange process. The final version featured seventeen categories and four levels. In addition, according to the expert advice the rubric was supplemented with source code examples, and categories were edited and put in a different order. Beyond that, a general "project characteristics" category with a keyword-like description of the project characteristics in the respective level was added as an orientation framework. Furthermore, a "creativity impression" category was introduced. Here, a subjective estimate of project creativity in the sense of technical originality and inventiveness is to be given.

4. Evaluation Process and Testing

To test the developed rubric, we prepared a dataset by selecting ten publicly available Snap! projects that reflect typical student work after their first exposure to programming. These projects varied in complexity, subject matter, and interactivity, ensuring a representative range of examples. This dataset illustrates the possible range of projects and serves as a reference for evaluation.

Finally, nine prospective teachers (male: 4, female: 5) majoring in computer science, technology, mathematics or natural sciences participated in the evaluation process. They had prior knowledge of Snap! or other block-based programming languages and possessed existing teaching experience. Initially, they rated the randomly sorted projects without any predetermined criteria using a school mark scale (1 = very good, A; 6 = insufficient, F). Afterward, they received the developed rubric and evaluated the same projects again based on the specified criteria. The evaluation of the competence grid was performed in German language.

4.1 Current version of the rubric

The current version of the rubric[i] comprises seventeen categories and four levels (0, 1, 2, and 3). For each category, a level can be awarded in one of the four levels. The overall level is determined as the sum of all points awarded within all categories.

The respective categories cover aspects of object-oriented development (e.g. objects or object communication), algorithmic design (use of loops, branches, reporters), handling of data (variables, lists), Snap! specific design options (graphic effects), handling of multithreading (header blocks and multithreading), and code outsourcing (BYOB). In addition, the "project characteristics" category describes a general implementation in relation to the corresponding level. The "creativity impression" category attempts to capture a subjective impression of the project that cannot be measured by the other categories. All categories and levels are listed below in descriptive statements translated into the English language.

1. Category “objects”

Level 0: Create an unstructured instruction sequence in a sprite.

Level 1: Create instruction sequences in an existing sprite to implement a specific function, e.g. object draws, object moves.

Level 2: Create and name another object(s) using a parallel statement sequence.

Level 3: Independently create several other objects with a communication or interaction for modelling a complex system.

2. Category “stage as an object”

Level 0: Cannot recognize stage as an object. No stage backgrounds/functions.

Level 1: Embed the stage in the system: set one or more backgrounds for the stage.

Level 2: Perceive the stage as an object: create a program for designing the stage, for example, by automatically changing the background images, using the graphic effects, time lapses.

Level 3: Perceive the stage as an object: create a program for the stage with object interaction.

3. Category “communication with a user AND/OR with other objects”

Level 0: Cannot use communication instructions.

Level 1: Use condition block to evaluate keyboard or mouse input or colour coding.

Level 2: Create simple communication between objects or with the environment.

Level 3: Create advanced communication between objects/with the user, for example via variables.

4. Category “Use of reporter blocks or predicates”

Level 0: Cannot demonstrate implementation of the reporter and predictor blocks.

Level 1: Use simple reporter blocks, such as random number or x-position.

Level 2: Use reporter/predicator blocks as parameter AND/OR in conditions.

Level 3: Use complex/composite reporter/predicator blocks.

5. Category “Graphical effects, sound effects, draw effects”

Level 0: Cannot demonstrate implementation of effects, etc.

Level 1: Use simple sound/speech/drawing instructions/graphical effects.

Level 2: Control the graphic effects AND/OR use combinations of different properties and sounds.

Level 3: Use graphical effects (effect combinations) meaningfully, for example to visualize a complex plot or to design the program interface.

6. Category “Hat blocks and multithreading”

Level 0: Always start instruction sequence without a hat block.

Level 1: Use a hat block to start the script, the script runs linearly.

Level 2: Create several scripts within a project, but without targeted use of the multithreading concept: scripts work independently of each other.

Level 3: Use several different hat blocks for a multithreading processing of the programs AND/OR use a hat block for sending the messages AND/OR "When I start as a clone".

7. Category “Object actions”

Level 0: Present a loose collection of instructions, no meaningful structure of a program.

Level 1: Create a sequence of instructions with fixed numerical values, e.g. with concrete size specifications AND/OR create a sequence of instructions for a sprite movement or figure geometry with waypoints.

Level 2: Use control flows with fixed values.

Level 3: Parameterize the statement sequence AND/OR use variables in control flows.

8. Category “Creating variables”

Level 0: Treat data as fixed values, with no variables present.

Level 1: Create and name a variable.

Level 2: Create several variables.

Level 3: Create (a) variable(s) for data exchange between objects (global variables) or within an object (local variables). Demonstrate meaningful use of local and global variables.

9. Category “Using variables”

Level 0: Use only numbers or words as constants.

Level 1: Change variables as numbers or strings in the course of the program.

Level 2: Change the value of a variable depending on a condition, for example, set false to true.

Level 3: Use variables as data containers for various data such as lists, objects.

10. Category “Using operators”

Level 0: Cannot show use of operators.

Level 1: Use simple mathematical operations, such as plus, minus, etc. in the function as a reporter.

Level 2: Use nested operators with variables AND/OR simple operators within a one-way branch/loop.

Level 3: Demonstrate meaningful use of complex operators, e.g. in conditions.

11. Category “Use of predicates in control flows”

Level 0: Cannot show existing termination condition (except for endless loop) AND/OR incorrect termination condition.

Level 1: Formulate a non-parameterized termination condition for a control flow.

Level 2: Formulate a parameterized termination condition for a control flow.

Level 3: Use operators (e.g. and, or, not) for a termination condition in a condition/loop AND/OR complex conditions (referring to other objects).

12. Category “Use of conditions”

Level 0: Cannot demonstrate implementation of conditions.

Level 1: Use an if-condition or an if-else condition.

Level 2: Use a nested branch AND/OR use a one-way branch for multiple cases.

Level 3: Show sensible use of complex nesting (but no unnecessary nesting, clear source code).

13. Category “Use of loops”

Level 0: Cannot demonstrate loops implementation.

Level 1: Use a loop.

Level 2: Use a combination of two loops (e.g. nesting them).

Level 3: Use multiple loops and complex loop structures, e.g. For loop.

14. Category “Use of lists”

Level 0: Cannot demonstrate list implementation.

Level 1: Create a simple list AND/OR output the list AND/OR prompt input for a list.

Level 2: Use list elements according to the respective index.

Level 3: Create lists with objects AND/OR further lists AND/OR use complex structures and commands.

15. Category “Build Your Own Block”

Level 0: Cannot demonstrate own block implementation.

Level 1: Combine several commands in their own blocks (outsource code).

Level 2: Create a block with a return value or with (a) parameter(s). Create reporter.

Level 3: Create a block with complex parameters AND/OR return values, such as lists and objects.

16. Category “Project characteristics”

(Selected examples; full description in the online version)

Level 0: A simple project with partly correct approaches but overall is inadequate or contains errors. No concept/no idea available. Loose collection of objects and functions.

Level 1: A project is manageable. 1-2 stage backgrounds are used. The plot is implemented with 2 to 3 objects. Simple control flows, instructions, operators are used.

Level 2: The project has a comprehensive structure. Several stage sets with effects are used. Control structures, instructions, links are used. Code is outsourced.

Level 3: The project has a complex structure. The plot is complex, exciting. Complex control structures, instructions, links, lists are used. Custom blocks are used with parameters and return values.

17. Category “Creativity impression”

Level 0: "The task has not been solved."

Level 1: "The task is solved, but not very creatively".

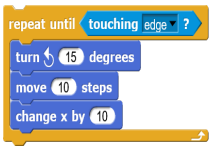
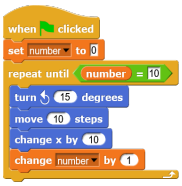
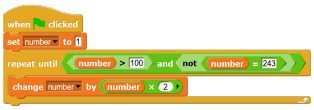
Level 2: "I understand the concept, it's exciting!"

Level 3: "Wow, that's a cool idea; a successful concept!"

Each level is described with a statement and, if useful, supplemented with a source code example (see example Tab. 1 "Using reporters and predicates"):

Table 1: Excerpt of a category from the rubric with four descriptions at each level and corresponding source code examples.

Category	Level 0	Level 1	Level 2	Level 3

Using predicates in control flows	Cannot show existing termination condition (except for endless loop) AND/OR use incorrect termination condition	Formulate a non-parameterized termination condition for a control flow	Formulate a non-parameterized termination condition for a control flow	Use operators (e.g. and, or, not) for a termination condition in a condition/loop AND/OR complex conditions (referring to other objects).
Code example	-			

4.2 Description of the test data set

The dataset used for the testing of the rubric consists of ten exemplary projects taken from the virtual Snap! library.

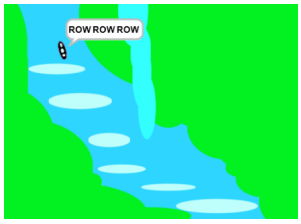
All of these projects can be found in the Snap! collection" at: The link has been hidden for the review process for anonymity reasons.

The following criteria were formulated for the dataset:

- The projects should, as far as possible, have different levels of complexity in source code, presentation, and plot.
- The projects can be interpreted as an average student performance after a teaching sequence in programming for novices.

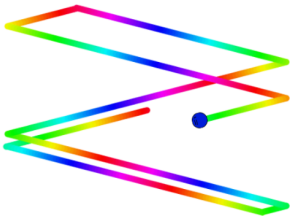
The following projects were selected. For reasons of clarity, the projects are presented in ascending order of complexity - Note: the test persons, however, received the projects in random order.

Project 1. Row row - Movement of an object along predefined waypoints.



The source code exhibits a linear, redundant structure. The object moves from one coordinate point to another, and the route is not automated. The outputs are implemented using a concurrently running script. The functionality and presentation are rudimentary. Overall, it is a simple project with some recognizable multi-threading usage.

Project 2: Rainbow Ball – Movement of an object along a random route with colour change.



The source code includes a loop and instructions from various categories such as movement, appearance, etc. The action is limited to visualization on the stage, and the representation is animated. Overall, it is an interesting project with an idea that was not further developed or implemented in a context.

Project 3: Exclusive Complexity – Calculating the average of 10 number inputs.

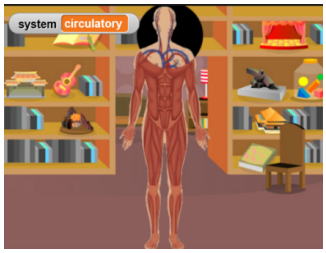
The source code has a linear structure, not parameterized, and lacks code modularization.

The program flow is linear, with a single thread of execution. The presentation includes a background image and an object. Overall, it is a simple project involving mathematical calculations.

Project 4: Avocado gif – An animated postcard featuring an avocado mother plant and its seed.

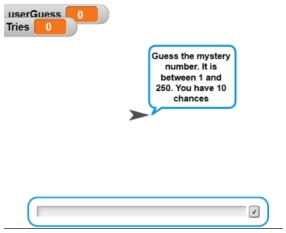
The source code implements multiple objects. Various control structures are used, and the code is modularized. The action runs concurrently. The action involves visualization without user interaction, and the representation is animated. Overall, it is a small but visually appealing project with a concept. The narrative flow could be further developed.

Project 5: Human body scanner – With a lens, various systems of the body can be observed.



The source code is concise. Instructions from different categories are utilized. There is no code modularization or user communication. Overall, it is a small, visually appealing project with potential for further enhancements.

Project 6: Guessing Game – The user is required to guess a number within a specific range.



The source code includes control structures and instructions for user communication. Code is modularized. There are no stage animations, only one object. Overall, it is a simple project related to a classic task.

Project 7: eCard Challenge – A game and an animated Halloween postcard combined into one. The user is required to answer quiz questions.

The source code incorporates various types of instructions. Object interaction and communication are present. The action and presentation are cohesive. However, the source code lacks modularization, resulting in redundant code segments. Overall, it is a good project with room for improvement.

Project 8: Fashion game – The user can dress and style a model.

An extensive project utilizing instructions from various categories, with code modularization. It has a complex structure, and the action and visualization complement each other. Overall, it is a comprehensive project with a clear concept.

Project 9: Dogder – A reaction game where the square object needs to avoid black dots.

The source code utilizes a comprehensive range of instructions and control structures. It involves complex interactions, a well-defined narrative flow, and efficient visualization. However, the code lacks modularization, resulting in some code redundancy and reduced readability. Overall, it is a complex project that showcases a wide range of functionalities. Project 10: Shooter Arcade Game – A shooting game with different levels of complexity.

An extensive project utilizing instructions from various categories, with efficient visualization. It has a complex structure, and the action and visualization complement each other. Overall, it is a comprehensive project with a clear concept, but the source code may be somewhat challenging to navigate due to its complexity

5. Results

Each project was first assessed with a school grade using German grading system from 1(A = very good) to 6 (F = insufficient). In the second part of the evaluation the test dataset was assessed with the described rubric.

The rubric consists of **17 categories**, each of which can be rated on four performance levels (0–3 points). This results in a **maximum attainable score of 51 points** (*17 categories* × *3 points*). To ensure comparability between the two evaluation phases, the total rubric score was converted into the German grading system using a linear transformation formula:

$$Grade = 6 - \frac{5 * RS}{MS}$$

RS: reached score; MS: maximum score

The following graph (Fig.2) illustrates the comparisons of the mean values of the ten assessments. The blue cross represents the average ratings of the projects using grades without the rubric, while the orange cross represents the mean rubric scores converted using the formula mentioned above.

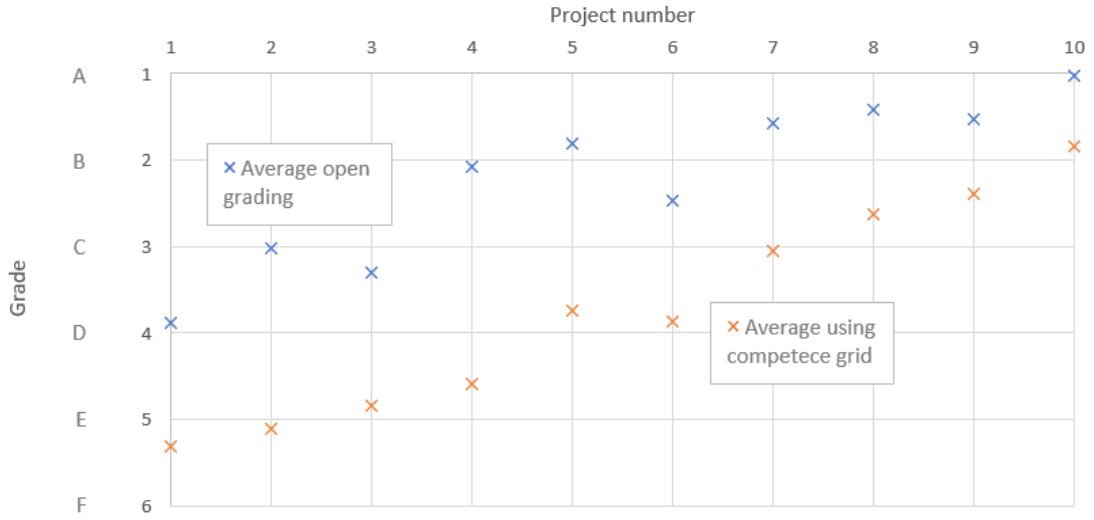


Figure 2: The average grades for test dataset. Blue crosses mark average results for grading without a rubric. Orange crosses mark the average results for grading using the rubric. For better comparison the score was converted in German school grades from 1(A) to 6(F)

It is noticeable that the projects assessed with the rubric receive significantly lower ratings. Even the best project, on average, achieves only a good grade (B) compared to grading without the rubric. Four projects do not meet the minimum requirements (grade D). In the open evaluation without a rubric, most projects achieve good to excellent grades. Both grading systems show in general the tendency from weak projects to good projects. Nevertheless, there are some outliers in the evaluation without a rubric, for example regarding projects 2 and 3 or projects 5 and 6. The difference in evaluation is indeed significant, with certain projects showing an average difference of two grades (e.g., Project 4).

The following diagram (Fig. 3) shows the individual results of the assessment without a rubric.

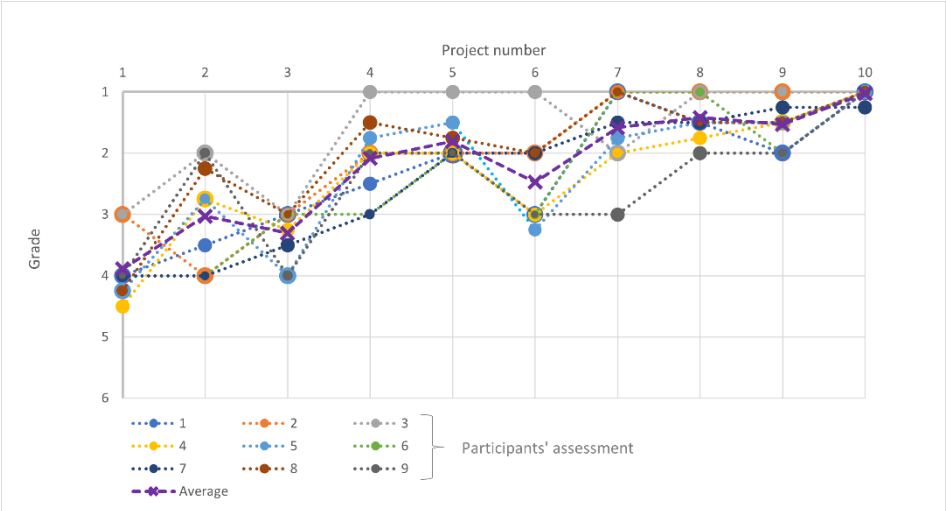


Figure 3: Dispersion of the results after assessing with a school grade with no predetermined criteria. Average is marked with a cross.

In the first part of subjective grade assessment of the projects, the spread of the grades for the respective projects is particularly striking. Project 2, for example, is assessed by the experts in a range of grades between 2 (good) and 4 (sufficient). The dispersion is strikingly high for all projects. Only project 10 is rated as a very good by all evaluators. Furthermore, it is striking that most projects tend to be assessed in the upper third of the rating range.

The assessment of projects shows wide variation within each individual evaluation. For example, one participant rates project 2 as well as projects 8 and 9 with a good grade, although in the direct comparison it remains questionable whether these three projects achieve the same level. At the same time, this participant rates projects 6 and 7 as significantly worse, with a satisfactory grade. Another participant also evaluates project 2 as good, but again evaluates projects 6,7,8 and 9 with an almost very good grade. In this evaluation model, it is thus not obvious according to which criteria the evaluations are made and a comparison of the evaluations among each other becomes almost impossible. Thus, this evaluation method does not appear to be transparent and cannot be used for the evaluation of the student projects.

When using the rubric, a higher consistency of the distribution of points can be observed (Fig. 4) and the results of the assessment show usually much lower dispersion.

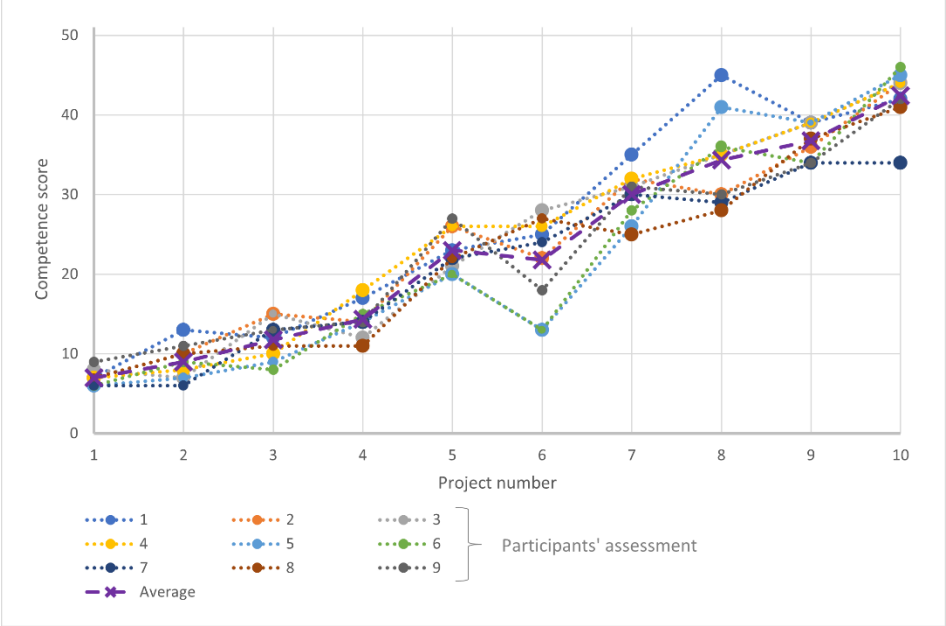
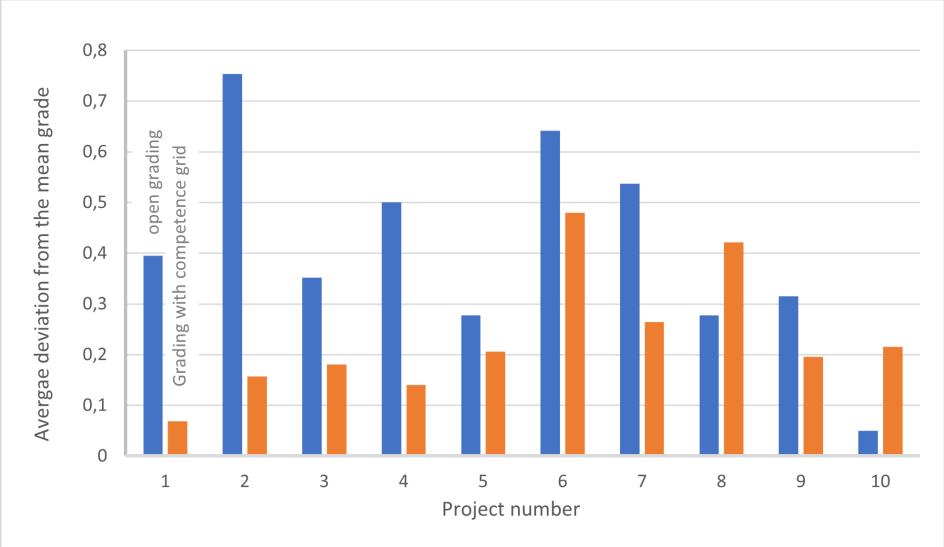


Figure 4: Dispersion of the results after assessing with a rubric. Maximum score result is 51 points for the best grade. Round dots mark the evaluation of German experts. Average is marked with a cross.

Clear outliers can be observed in projects 6 and 8. In project 6, code outsourcing (Category 15: Build Your Own Block) in own blocks was sometimes overlooked during the assessment, resulting in an incorrect assessment. Project 8 was partially classified as having a too high level of mastery. Presumably, differences

between the second and third levels of the rubric can be recognized less easily by inexperienced raters. For example, for this project, the third level is awarded in the categories on loops and branches, even though the source code has a level of only two. Despite the observed inconsistencies, the most projects can be assessed more homogeneously in each case. All ratings are mostly within one grade. This means that projects can be better assigned to the different levels. Thus, the projects are evenly distributed among the lower, middle, and high score ranges. The following graph (Fig. 5) shows the average deviation in grade points from the mean assessment grades for the respective projects with and without rubric. The overall mean deviation is 0.41 without the rubric. Without the ceiling effect, the deviation would probably be significantly higher, especially for good projects (8-10 comparable with projects 1-7). The overall mean deviation is 0.24 with the rubric, demonstrating a substantial decrease in rating variability and improved assessment consistency.

Figure 5: Average deviation from mean grade, blue colour for grading without rubric and orange colour for grading with the rubric.



The dispersion of values around the mean is significantly smaller by using the rubric. The highest average deviation is 0.48 grade points. In contrast, without the rubric, the maximum average deviation from the mean is 0.75 grade points. However, particularly as project complexity increases, the dispersion in evaluations with the rubric also tends to increase. There could be two reasons for this. First, evaluating complex structures requires more knowledge in assessment, making it more challenging for inexperienced evaluators. It may indicate inexperienced assessors´ inability to differentially assess complexity, as well as their tendency to uniformly assign a good grade. Second, a smaller evaluation dispersion without the rubric does not necessarily indicate a better quality of assessment. Rather, as the note scale stops, a ceiling effect occurs. The assessment results show a ceiling effect, where many projects are concentrated at the higher end of the score range, making it difficult to distinguish between them. It seems, project 10 is assessed as attaining the highest level by almost all evaluators without a rubric. Obviously, this project works as a standard in comparison to

other projects because it is the most complicated example. That is the probable explanation for the highest score on the open grading. But if the projects are mapped to a rubric standard, project 10 does not achieve the best possible grade because it does not fulfil all the requirements. This project, like project 8, has a complicated structure, so evaluators probably have difficulties scoring it, even with a rubric. Therefore, this could explain higher dispersion in the evaluation of complicated projects. The second reason could be that at higher levels; the rubric allows for more room for interpretation and evaluative freedom. Overall, it can be still said that in most projects (except projects 8 and 10), the average deviation from the group mean is significantly smaller when using the framework, as Fig. 5 clearly shows.

An individual comparison of the ratings by example person (light grey dot) is shown in the following diagram (Fig. 6). This is a participant who awards a very different rating, both with and without the rubric. The participant's ratings without the rubric revealed a ceiling effect, as most projects were scored highly, often receiving A grades. In contrast, when using the rubric, their evaluations became more nuanced and differentiated, indicating a more refined assessment of the projects.

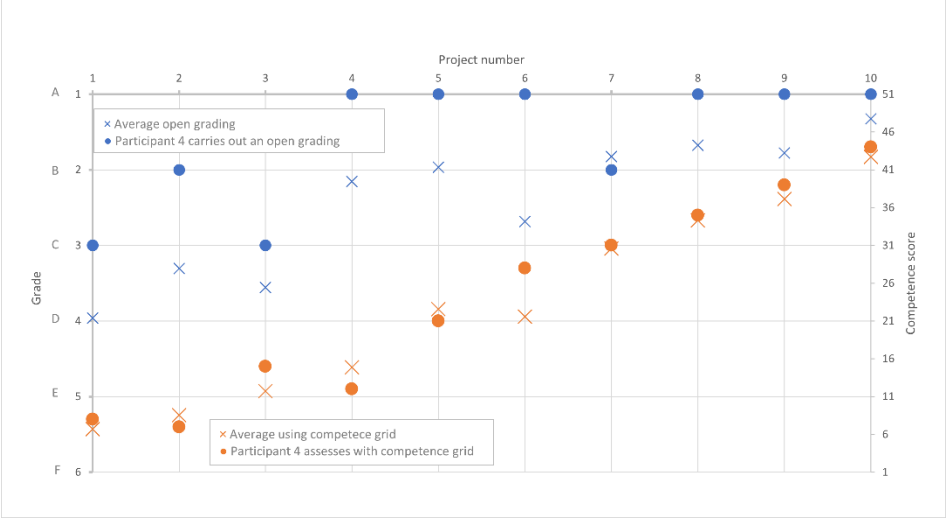


Figure 6: The results for a single example participant. Orange dots mark the evaluation with a rubric; blue dots mark the evaluation without a rubric. Corresponding group average is marked with crosses.

The example data set shows the effects of the rubric. Originally, this participant rated projects significantly higher than the average. For example, project 4 deteriorates from a very good (1, A) grade to a "fail" (5, E). Projects 1, 3, 5, 6, 8 also experience a significant deterioration. This is an interesting phenomenon that could not be clarified within the framework of this evaluation. There was a tendency for all evaluators to be significantly higher at open grading. Furthermore, when the results of example person are compared with the mean values, it is noticeable that, with the rubric, the assessment is closer to the general mean values. The individual mean deviation is 0.04 grading points (2.12 of 51 points). Except for one project (7), the results of open grading by example person are far from the average grading. The mean deviation with open grading is 0.86 grading points. The rubric allows each rater to evaluate using the same scale as all other raters.

To examine the consistency of the evaluations and to determine whether the rubric improved the objectivity of grading, the interrater reliability (IRR) was calculated for both evaluation phases. For the free grading phase without predetermined criteria, Kendall's coefficient of concordance (W) was applied, as this method is appropriate for ordinal data such as school grades (Gibbons, 1993; Olson et al., 2003; Venugopal et al., 2024). For the rubric-based evaluation, the raw scores were first converted into the German grading system with increments of 0.25 (e.g., 5.81 → 5.75) to ensure a direct comparison with the free grading phase. A higher Kendall's W indicates greater agreement among raters, with values ranging from 0 (no agreement) to 1 (perfect agreement) (Olson et al., 2003).

This quantitative analysis provides an objective measure of the reliability of the evaluation process and demonstrates whether the rubric successfully reduced subjective variation in grading. The following section first presents **the Kendall's W values** for both evaluation phases and compares the level of agreement among raters.

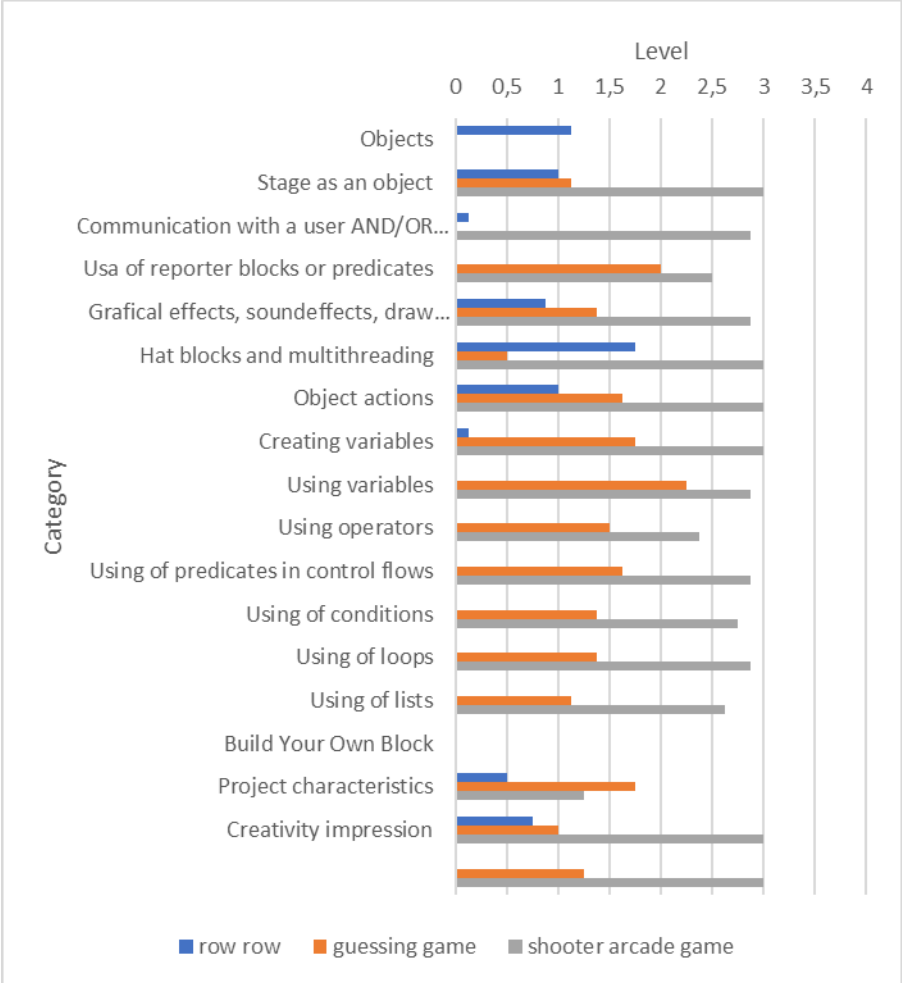


Figure 6: Comparison of results for individual projects by category. Selection of three projects. The picture shows an average of all graders.

The interrater reliability analysis revealed a clear difference between the two evaluation phases. Without the rubric, the agreement among the nine raters was **moderate to good**, with a Kendall's W of **0.634** ($p < .001$). When using the developed rubric, the level of agreement increased substantially to **W = 0.940** ($p < .001$), indicating **very high to almost perfect concordance** between raters.

This result demonstrates that the rubric not only provides a structured framework for evaluation but also significantly reduces subjective variation in grading. The substantial increase in Kendall's W suggests that the competence grid helped the raters to apply more consistent and comparable evaluation criteria, thus improving the reliability of the assessment process.

Moreover, the designed rubric enables a more refined analysis of project quality at the individual level, providing a detailed breakdown of each project's strengths and weaknesses. Fig. 7 illustrates this capability, presenting a comparative analysis of three exemplary projects, highlighting their distinct characteristics and achievements. This nuanced evaluation allows educators to provide targeted feedback, fostering growth and improvement in each student's programming skills.

This representation method can help to break down each assessment individually into strengths and weaknesses as needed. Using the evaluation results, it is possible to explicate single components and compare them. Specifically in this example, the selected sample dataset could be reviewed in terms of existing concepts and the level of proficiency achieved on average. For example, in the category Use of conditions, the "shooter arcade game"- project achieves a high level of mastery, "guessing game" project shows moderate expertise, and the "row row" project lacks understanding in this area. On the base of this analysis method, it is possible to evaluate learning goals and correlated results and to give more detailed feedback on each project.

6. Discussion and Conclusion

The results of this study demonstrate that the rubric, with its 17 categories, is a comprehensive tool for evaluating programming projects. The primary aim of creating a structured description for creative block-based programming projects was successfully addressed with this rubric, providing a clear and systematic framework for assessment. Statistical analysis confirmed the reliability of the rubric. Kendall's W showed a high degree of agreement between assessors, demonstrating that the rubric supports consistent assessments by different assessors. At the same time, the distribution of scores indicated a possible ceiling effect, as particularly good projects achieved the highest possible score in several categories. This result shows that future iterations of the rubric could benefit from the addition of more advanced descriptions in order to better distinguish particularly high-performing projects. The lack of correlation between categories suggests that each category provides unique insights into the project's quality. As a result, the number of categories cannot be reduced without compromising the effectiveness of the evaluation when the rubric is used to derive a grade. However, for purely qualitative evaluations, certain categories may be excluded, particularly when specific

aspects have not been covered during instruction.

The comparison of the two assessment forms clearly demonstrated that using a rubric led to criterion-led assessment, significantly reducing the average deviation of grades and thereby improving comparability between evaluators. Qualitative feedback from the raters confirmed their satisfaction with the tool, highlighting its clarity, perceived objectivity, and the sense of "clear conscience" when grading. The included source code examples were particularly valued, especially by less experienced assessors.

In terms of feasibility, evaluating programming projects with the rubric proved manageable. Assessing a single project required about nine minutes, totalling roughly 270 minutes for a class of 30 students. While no empirical data exist for grading times in computer science, this workload is comparable to grading a standard mathematics test, which typically takes around 360 minutes (Frank et al., 2023) . Thus, the rubric is not only reliable and comprehensive but also practical for classroom use, even in larger cohorts.

Nevertheless, the rubric has limitations. Its construction is based on student projects and qualitative expert assessment. As a result, not all possible Snap! categories are currently covered in the rubric, indicating a need for ongoing research and refinement. There is room for further differentiation of category descriptions, and higher proficiency levels would benefit from additional examples to support the application of the rubric.

Implementing the rubric in diverse educational contexts may pose challenges, particularly when teachers have limited experience with programming and assessment. The results of this study indicate that the rubric can be especially useful in such cases, as it helps to harmonize evaluations and align them with the mean value, as illustrated in Figure 6. These findings highlight the potential of the rubric to support less experienced teachers and suggest that future research should explore strategies to further facilitate its effective use.

Currently, there is no standardized, empirically validated framework for the evaluation of block-based programming projects. Existing approaches vary widely and are often designed for standardized, task-based contexts rather than authentic, open-ended projects. This study contributes to filling this gap by providing a structured, qualitative instrument for assessing Snap! projects, while also laying the groundwork for future comparative studies and broader validation efforts.

The developed rubric may also be applicable to other block-based programming languages such as Scratch. However, this potential transferability was not examined within the scope of the present study. Future research should therefore investigate its suitability across different programming environments to validate and possibly extend its applicability. From the students' perspective, the rubric can also serve as a reference framework to understand expectations and support self-assessment. Finally, by providing clear, criterion-based guidance, the rubric helps to overcome common challenges in evaluating problem-solving skills within the computational thinking process, particularly in the areas of algorithmic design, parallelization, iteration, and automation.

References

Andrade, H. G. (2000). *Using Rubrics to Promote Thinking and Learning*.

Balouktsis, I. (2016). Learning Renewable Energy by Scratch Programming. *Επιστημονική Επετηρίδα Παιδαγωγικού Τμήματος Νηπιαγωγών Πανεπιστημίου Ιωαννίνων*, *9*(1), 129. https://doi.org/10.12681/jret.8916

Bau, D., Gray, J., Kelleher, C., Sheldon, J., & Turbak, F. (2017). Learnable programming: Blocks and beyond. *Communications of the ACM*, *60*(6), 72–80. https://doi.org/10.1145/3015455

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., & Franklin, D. (2013). Hairball: Lint-inspired static analysis of scratch projects. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 215–220. https://doi.org/10.1145/2445196.2445265

Brennan, K., & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking*.

Da Cruz Alves, N., Gresse von Wangenheim, C., & Hauck, J. C. R. (2019). Approaches to Assess Computational Thinking Competences Based on Code Analysis in K-12 Education: A Systematic Mapping Study. *Informatics in Education*, *18*, 17–39. https://doi.org/10.15388/infedu.2019.02

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. https://doi.org/10.1016/j.compedu.2011.08.006

Döring, N., & Bortz, J. (2016). *Forschungsmethoden und Evaluation in den Sozial- und Humanwissenschaften*. Springer Berlin, Heidelberg. https://doi.org/10.1007%2F978-3-642-41089-5

Frank, M., Thomas, H., & Martin, R. (2023). *Arbeitszeit und Arbeitsbelastung von Lehrkräften an Schulen in Sachsen 2022: Ergebnisbericht*. https://doi.org/10.47952/gro-publ-172

Funke, A., & Geldreich, K. (2017). Measurement and Visualization of Programming Processes of Primary School Students in Scratch. *Proceedings of the 12th Workshop on Primary and Secondary Computing Education - WiPSCE '17*, 101–102. https://doi.org/10.1145/3137065.3137086

Garcia, D. D., Harvey, B., & Segars, L. (2012). CS principles pilot at University of California, Berkeley. *ACM Inroads*, *3*(2), 58. https://doi.org/10.1145/2189835.2189853

Gesellschaft für Informatik (Hrsg.). (2016). Bildungsstandards Informatik—Sekundarstufe II. *Empfehlungen der Gesellschaft für Informatik e. V. erarbeitet vom Arbeitskreis »Bildungsstandards SII«*, *183/184*, 88.

Gibbons, J. (1993). *Nonparametric Measures of Association*. SAGE Publications, Inc. https://doi.org/10.4135/9781412985291

Gummels, I. (2020). *Wie kooperatives Lernen im inklusiven Unterricht gelingt*. Springer Spektrum Wiesbaden. https://doi.org/10.1007/978-3-658-29114-3

Harel, I., Massachusetts Institute of Technology, & Media Laboratory (Hrsg.). (1993). *Constructionism: Research reports and essays, 1985-1990* (2. print). Ablex Publ. Corp.

Hattie, J. (2009). *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Routledge.

Jürgens, E., & Lissmann, U. (2015). *Pädagogische Diagnostik*. Beltz Verlag.

Koh, K. H., Basawapatna, A., Nickerson, H., & Repenning, A. (2014). Real Time Assessment of Computational Thinking. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 49–52. https://doi.org/10.1109/VLHCC.2014.6883021

Koray, A., & Bilgin, E. (2023). The Effect of Block Coding (Scratch) Activities Integrated into the 5E Learning Model in Science Teaching on Students' Computational Thinking Skills and Programming Self-Efficacy. *Science Insights Education Frontiers*, *18*(1), 2825–2845. https://doi.org/10.15354/sief.23.or410

Krugel, J., & Ruf, A. (2020). *Learners' perspectives on block-based programming environments: Code.org vs. Scratch*. http://doi.acm.org/10.1145/3421590.3421615

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, *10*(4), 1–15. https://doi.org/10.1145/1868358.1868363

Mladenović, M., Mladenović, S., & Žanko, Ž. (2020). Impact of used programming language for K-12 students' understanding of the loop concept. *International Journal of Technology Enhanced Learning*, *12*(1), 79. https://doi.org/10.1504/IJTEL.2020.103817

Modrow, E. (2018). *Informatik mit Snap!, Snap! In Beispielen*. http://ddi-mod.uni-goettingen.de/InformatikMitSnap.pdf

Moreno-León, J., Román-González, M., Harteveld, C., & Robles, G. (2017). On the Automatic Assessment of Computational Thinking Skills: A Comparison with Human Experts. *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2788–2795. https://doi.org/10.1145/3027063.3053216

Olson, L., Schieve, A. D., Ruit, K. G., & Vari, R. C. (2003). Measuring Inter-rater Reliability of the Sequenced Performance Inventory and Reflective Assessment of Learning (SPIRAL): *Academic Medicine*, *78*(8), 844–850. https://doi.org/10.1097/00001888-200308000-00021

Papavlasopoulou, S., Giannakos, M. N., & Jaccheri, L. (2019). Exploring children's learning experience in constructionism-based coding activities through design-based research. *Computers in Human Behavior*, *99*, 415–427. https://doi.org/10.1016/j.chb.2019.01.008

Papert, S. (1993). *The children's machine: Rethinking school in the age of the computer*. BasicBooks.

Perera, P., Tennakoon, G., Ahangama, S., Panditharathna, R., & Chathuranga, B. (2021). A Systematic Mapping of Introductory Programming Languages for Novice Learners. *IEEE Access*, *9*, 88121–88136. https://doi.org/10.1109/ACCESS.2021.3089560

Price, T. W., & Barnes, T. (2015). Comparing Textual and Block Interfaces in a Novice Programming Environment. *Comparing Textual and Block Interfaces in a Novice Programming Environment*. https://doi.org/10.1145/2787622.2787712

Resnick, M. (2014). *Give P's a chance: Projects, peers, passion, play*.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. https://doi.org/10.1145/1592761.1592779

Resnick, M., Silverman, B., Kafai, Y., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., & Silver, J. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60. https://doi.org/10.1145/1592761.1592779

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 59–66. https://doi.org/10.1145/2493394.2493403

Shute, V. J. (2008). Focus on Formative Feedback. *Review of Educational Research*, *78*(1), 153–189. https://doi.org/10.3102/0034654307313795

Svedkijs, A., Knemeyer, J.-P., & Marmé, N. (2022). Förderung von Computational Thinking durch ein digitales Leitprogramm zur blockbasierten Programmiersprache Snap! In B. Stadl (Hrsg.), *Digitale Lehre nachhaltig gestalten*. Waxmann Verlag. https://doi.org/10.31244/9783830996330

Tsai, C.-Y. (2019). Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior*, *95*, 224–232. https://doi.org/10.1016/j.chb.2018.11.038

Venugopal, V., Dongre, A., & Kagne, R. N. (2024). Development of an analytical rubric and estimation of its validity and inter-rater reliability for assessing reflective narrations. *The National Medical Journal of India*, *36*, 323–326. https://doi.org/10.25259/NMJI_732_21

Weintrop, D., & Wilensky, U. (2015). To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. *Proceedings of the 14th International Conference on Interaction Design and Children*, 199–208. https://doi.org/10.1145/2771839.2771860

Weintrop, D., & Wilensky, U. (2017). Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, *18*(1), 3.

Weintrop, D., & Wilensky, U. (2018). Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education*, *18*(1), 1–25. https://doi.org/10.1145/3089799

Wen, F.-H., Wu, T., & Hsu, W.-C. (2023). Toward improving student motivation and performance in introductory programming learning by Scratch: The role of achievement emotions. *Science Progress*, *106*(4), 00368504231205985. https://doi.org/10.1177/00368504231205985

Werner, L., Denner, J., Campe, S., & Kawamoto, D. C. (2012). The fairy performance assessment: Measuring computational thinking in middle school. *ACM Transactions on Computing Education*, 215–220. https://dl.acm.org/doi/10.1145/2157136.2157200

Wiliam, D. (2011). *Embedded formative assessment*. Solution Tree Press.

Wolf, K., & Stevens, E. (2007). The Role of Rubrics in Advancing and Assessing Student Learning. *The Journal of Effective Teaching*, *7*(1), 3–14.

Zhang, N., & Biswas, G. (2019). Defining and Assessing Students' Computational Thinking in a Learning by Modeling Environment. In S.-C. Kong & H. Abelson (Hrsg.), *Computational Thinking Education* (S. 203–221). Springer Singapore. https://doi.org/10.1007/978-981-13-6528-7_12

[i] https://www.innovation-tank.de/teaching/ - Rubric available for download.