Block-Based Programming for Mobile with Conventional Exceptions and Automatic Evaluation

Aryobarzan Atashpendar
Department of Computer Science
University of Luxembourg
Esch-sur-Alzette, Luxembourg
aryobarzan.atashpendar@uni.lu

Steffen Rothkugel
Department of Computer Science
University of Luxembourg
Esch-sur-Alzette, Luxembourg
steffen.rothkugel@uni.lu

ABSTRACT

Block-based programming has been used as an introductory gateway to programming thanks to its simpler, visual approach for composing code. This work re-imagines this style of coding through the mobile-first framework DartBlock, which is intended to be integrated in a quiz application. It targets an older audience aiming to more quickly transition to the conventional programming language Java, by adopting a syntax visualization and interaction design inspired by integrated development environments. DartBlock focuses on teaching debugging through its included runtime which allows the execution of faulty code with descriptive exception throwing, as well as supporting automatic evaluation of programs given a sample solution to serve as feedback. To suit the smaller screen of mobile devices, the library relies on calculator-like visual editors for each primitive data type to facilitate the composition of complex boolean and numeric expressions without the need for textual input. Finally, we present preliminary student impressions of DartBlock from an initial use case study in a university class and provide organizational recommendations for introducing new frameworks to students.

CCS CONCEPTS

• **Applied computing** → **E-learning**; • **Human-centered computing** → **Interaction design**.

KEYWORDS

Block-based Programming, Automatic Evaluation, Digital Quiz

ACM Reference Format:
Aryobarzan Atashpendar and Steffen Rothkugel. 2024. Block-Based Programming for Mobile with Conventional Exceptions and Automatic Evaluation. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653549

1 INTRODUCTION

Basic IT programming skills have become a prevalent requirement for disciplines other than computer science: researchers need to use advanced statistical tools provided by the Python and R languages, user interface designers want to create prototypes for their concepts, and so on. In turn, new learning methods are needed to teach these skills at a starter level. Concepts such as variables, functions, types, decision structures and loops need to be learned, which leads to a complex set of learning outcomes for students who are not necessarily from a computer science background.

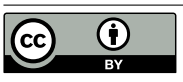
To that end, a novel approach to learn programming has been the usage of block-based programming [9, 20], which entails building programs via visual interaction rather than typing: the user drag-and-drops blocks from a toolbox and relies on their shapes and color-coding [13, 14, 21] to form a syntactically correct statement, similar to puzzle pieces. The idea is to relieve the user of learning the syntax of a given programming language by heart [23], as block-based coding frameworks not only provide the available blocks, but they are designed to prevent the student from building syntactically incorrect programs [9], as the block shapes indicate which components go together. Similarly, the concept of variable scopes is conveyed via the nesting of blocks, while as a final step newer frameworks also allow the user to export their block-based program to a conventional programming language [7, 10], such as Java. The latter is seen as a crucial step to ease the learner's transition from block-based to conventional programming.

Block-based tools have been used with learners of various ages to assess their effectiveness. Participants have noted that they could learn the building blocks of programming [15], such as variables and conditional logic, as well as general algorithmic thinking skills [11].

Younger learners also appreciated the ability to tackle mathematical concepts [12] such as decimals and negative numbers which they find easier to challenge through interactive, block-based tools.

While there have been studies showing that block-based tools enabled students to achieve higher learning outcomes compared to text-based learners [23], others have not shown a significant difference in the learning outcome itself [19], but rather in the speed of knowledge acquisition, where it took less time for block-based learners to learn the programming concepts.

However, this brings up the shortcomings of block-based programming [23], some of which are based on the perception of students: firstly, block-based coding can be slower compared to typing, due to having to drag-and-drop different blocks to form a single statement, whereas students with knowledge of the text-based syntax can quickly type out the equivalent code. Similarly, some students perceive typed code as more concise and easier to understand: block-based frameworks become cumbersome to use when scaling up to larger programs [3], resulting in a lot of blocks, whereas the equivalent typed code can be both shorter and easier to grasp an overview of. Additionally, composing more complex



ITiCSE 2024, July 8–10, 2024, Milan, Italy

ACM ISBN 979-8-4007-0600-4/24/07
https://doi.org/10.1145/3649217.3653549

expressions such as boolean conditions can be more inconvenient and take longer compared to typing them out.

Furthermore, students have bemoaned the "authenticity" [23] of block-based code, i.e., they believe it would be more useful to directly learn coding through conventional programming languages, as they believe the difference between block-based code and the syntax of written code to be too stark to result in an easy transfer of knowledge when making the move to traditional programming.

Lastly, learning programming through block-based frameworks can also lead to bad habits [16], such as writing too fine-grained code when transitioning to conventional programming and using inappropriate structures (decision structures, loops) in cases where other structures would achieve the same outcome more efficiently.

Our work focuses on a new block-based programming framework, referred to as DartBlock, designed as both a standalone editor as well as a module to be integrated in other apps. The goal is to more quickly transition learners to conventional coding, while also supporting features that facilitate its inclusion in a quiz app. Firstly, DartBlock adopts a mobile-first design, with the user interface (UI) and interaction being modelled for the Android and iOS platforms. A touch-heavy interaction reduces the need for typing and provides a visual approach to composing complex expressions which can consist of constants, variables and function calls.

Furthermore, while DartBlock relies on a toolbox with available "blocks" to drag-and-drop to the canvas, it adopts a more conventional representation of these elements: the blocks are modelled as entire statements, which are visualized as individual lines, similar to conventional code, thus breaking away from the concept of placing multiple blocks together to form a single statement. This approach can ease the transition to typed-out code as the representation already resembles it closely, as well as potentially reduce the number of gestures required to form statements. However, DartBlock still relies on color-coding, labels and indentation to improve readability of the code and to convey information about variable scopes.

Next, to better teach debugging, DartBlock allows the user to compose both syntactically and semantically malformed programs, e.g., by assigning the wrong value type to a variable or causing an infinite loop, something that other frameworks such as Scratch prevent to avoid the need for users to know the underlying syntax. This enables DartBlock to simulate exceptions being thrown, e.g., stack overflow exceptions, which replicate the experience of running written code in an integrated development environment (IDE), i.e., DartBlock describes the type of exception being thrown and the statement causing it.

Finally, DartBlock was developed to be integrated as a new question format in a mobile quiz application [1], whose objective is to support regular testing by students to enhance their long-term retention, where the effect of the testing is positively affected by the presence of immediate feedback to the user's answer. For that purpose, DartBlock provides different schemas of automatic evaluation, such that the quiz app can evaluate the user's constructed program against the teacher's sample solution. Additionally, as the quiz app adapts the difficulty of questions to each user's level, DartBlock provides levers for adjusting a program's difficulty, such as by randomly trimming parts of it, shuffling the statements, or even generating hints.

Though in its early stages, DartBlock was tested as part of a quiz app in a university-level programming class during a semester, for which we present preliminary results.

The remainder of this paper is organized as follows. In Section 2, we highlight existing block-based coding frameworks. Next, we describe the design and feature-set of our proposed DartBlock package in Section 3, as well as the findings of our initial case study in Section 4. We discuss the current state of DartBlock in Section 5 and conclude in Section 6 with the planned future work.

2 RELATED WORK

Scratch [14] is a prominent visual programming framework for young learners to learn programming without any prior experience. It adopts many of the hallmark features of block-based coding such as color-coding and puzzle-like block shapes to simplify the composition of syntactically correct code. However, Scratch masks certain aspects of conventional programming to not overload the user: firstly, while it supports the boolean, number and string types, its variables are untyped. Additionally, while runtime errors are possible, e.g., due to a division by zero, Scratch simply highlights the error-causing block through a red border without an error message. Scratch also does not allow a total failure in the case of a runtime error, but it relies on "failsoft" blocks, e.g., if the user is using an out-of-bounds index for a list element lookup, Scratch automatically bounds the index to a valid value. Scratch thus attempts to teach the basics of programming while hiding technical details related to language-specific syntax, typing and error handling. However, as it targets younger students, it incorporates 2D sprites and audio to allow the learner to create games and to have their program's execution include a concrete visualization, both to drive engagement and to better illustrate each block's functionality.

On the other hand, the Blockly [18] library does not support the execution of programs. Rather, developers are not only responsible for building the runtime engine, but also adapting the various aspects of Blockly to their own use case, e.g., whether blocks should be represented using natural language or programming-specific language depending on the target audience. The main appeal of Blockly is its open source and extensible nature, as well as the ability to integrate it into websites and mobile applications as a WebView component. Moreover, one of its notable features is the option to export block-based programs to specific programming languages such as Javascript, which can aid in the learner's transition to conventional programming.

Next, the Alice [4, 5, 7, 17] software aims to teach programming through a graphical approach to an older audience. It relies on similar manipulation methods, such as choosing blocks from a toolbox and preventing combinations of blocks which would be syntactically incorrect, but Alice also supports object-oriented concepts such as inheritance and polymorphism, which are absent in Scratch. Alice allows the user to execute their program to better analyze the behavior of their blocks, though in this case the execution involves 3D animations of objects as opposed to the 2D manipulation of sprites in Scratch. Similar to Blockly, Alice can export programs to a conventional text-based language, though it limits itself to Java as an option.

Furthermore, there is the hybrid approach of Pencil Code [2, 8], where the user can switch between block-based code and textual code. In the latter case, they can also directly manipulate the text-based code by typing. Pencil Code mainly targets older students by exposing them to the syntax of a conventional language such as Javascript, where they can also edit said code through typing rather than just through visual interaction with abstract blocks.

Lastly, the mobile-focused TouchDevelop [22] envisioned a visual programming style targeting the Windows Phone with many of our work's goals, such as a runtime to execute programs on-device, a visualization close to text-based code and a calculator-like interface for building expressions. However, TouchDevelop deliberately does not feature error handling to simplify programming for beginners, nor does it include automatic evaluation or difficulty adjustment in the context of quizzes. Additionally, while it provides a keypad to ease the composition of expressions, TouchDevelop internally models expressions as free form text which has to be parsed before being evaluated, i.e., the user can compose both semantically and syntactically incorrect expressions.

3 DARTBLOCK

3.1 Technical Implementation

DartBlock was implemented using the Dart[1] language and the related cross-platform framework Flutter[2]. The reason was to use a mobile-centric toolset to develop an Android and iOS friendly interface, as illustrated in Figure 1, including both the look and the touch-heavy interaction. Additionally, Flutter allows the deployment of DartBlock to every other major platform, including macOS, Windows, Linux and the Web, making it easily re-usable even outside mobile platforms. Support for the Web platform means the DartBlock editor can even be embedded into websites as an HTML component, including with Javascript callbacks for cross-communication. As DartBlock has been developed as a Dart package, it can be natively integrated into any Flutter app.

3.2 Statements

DartBlock's initial version features the following building blocks, otherwise referred to as statements:

- Declare Variable (integer, double, boolean, String)
- Update Variable (assign a new value to a variable)
- For-Loop, While-Loop, Break, Continue (loops)
- If-Then-Else (decision structure)
- Print (to console)
- Return Value
- Call Function

The last two statements allude to custom functions, where the user can declare both the signature (function name, return type, parameters list) and the body of the function which is a list of statements. To help the user, DartBlock can also throw exceptions when the user runs a program containing a custom function call, e.g., which is faulty due to a lack of a "Return Value" statement in the custom function's body when its return type is not void.

[1]Dart: programming language (dart.dev)
[2]Flutter: cross-platform app development framework (flutter.dev)

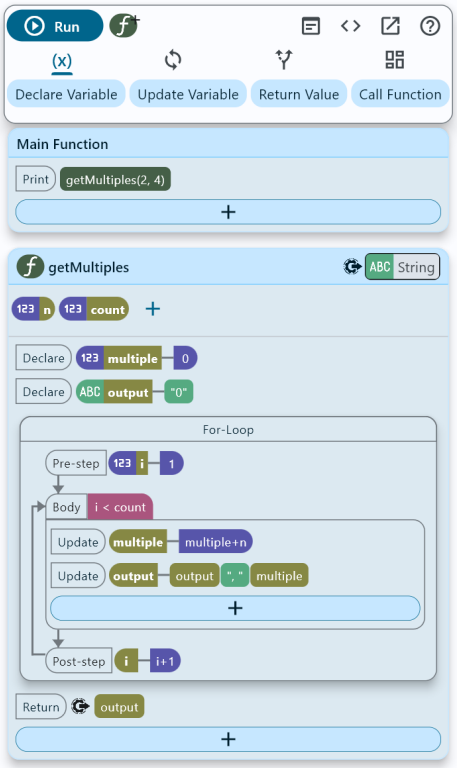


Figure 1: DartBlock's interface with the toolbox at the top.

As DartBlock models itself after the Java language for our prototype, the available primitive types and general syntax also imitate the equivalent counterparts from Java. This also includes the truncating behavior of Java when it comes to integer division, which differs from the behavior of integer division in the Dart language, hence it has been specifically overridden for this context.

Furthermore, compared to Scratch's untyped approach [14], DartBlock requires the user to indicate the type of a new variable when declaring it such that they learn about the typed nature of Java. Similarly, while Scratch allows the user to drop a block onto the canvas which it automatically sets up with default values, DartBlock always prompts the user for the required parameters and values of a given statement being drag-and-dropped onto the canvas: for example, the "Declare Variable" statement will ask the user to indicate the type and name of the variable, as well as optionally an initial value, similar to how variable declarations are typically done in Java. However, the user is not always required to manually type out certain required parameters, e.g., when adding a "Update Variable" statement, DartBlock gives the user a menu from which they can pick an existing variable they want to update based on the current scope, rather than typing out the variable's name.

Nevertheless, one statement for which DartBlock provides default values upon creation is the "For-Loop", due to its more complex syntax owing to its different steps. To help beginners, DartBlock populates new for-loops as `"int i = 0; i < 10; i++"` (Java syntax), which the user can then edit to their liking, such as replacing the pre-execution condition.

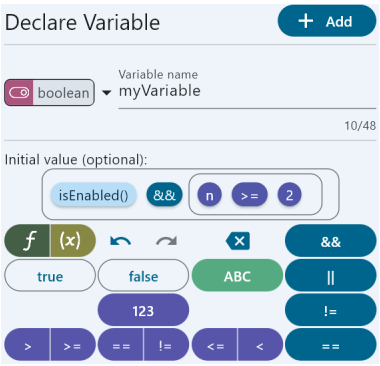


Figure 2: Declaring a boolean variable using a calculator-like interface requiring no manual typing for the value.

3.3 Expression Composer

To facilitate the composition of longer boolean, numeric or String expressions, DartBlock relies on a calculator-inspired interface for each expression type to reduce the need for manual typing.

The simplest editor is the numeric composer, which is similar to a calculator, though it includes two extra buttons for also choosing variables or creating function calls to be included in the expression.

The boolean composer, shown in Figure 2, in turn provides buttons for the constant values true and false, as well as the basic comparison operators (and, or, equal, not equal). Additionally, it provides dedicated, color-coded buttons for composing numeric comparison expressions (greater than, less than, etc.), which in turn re-use the numeric composer by opening it as a pop-up window. Finally, the String composer allows for the concatenation of any type of value, including constant values (String, boolean, number), variables and function calls. The numeric and boolean composers are re-used here when the user wants to append a numeric or boolean expression to their String.

3.4 Syntax

DartBlock relies on color-coding to visualize the different types of UI elements, though its representation of the statements is more akin to written code than block shapes, i.e., statements are displayed as individual lines with basic descriptors ("Declare", "For-Loop", etc.) and indentation is used to visualize scopes, e.g., of nested loops. However, DartBlock uses a more bespoke design for for-loops, where the various components (pre-initialization step, condition, body, post-execution step) are clearly distinguished in the UI to better teach the different steps and their order of execution. Additionally, while TouchDevelop [22] relies on pagination, DartBlock keeps the canvas visible in the background while statement editors are shown as floating views on top of it. The goal is to not confuse the user with multiple levels of pagination while editing statements.

3.5 Scopes & Recursion

DartBlock supports variable scopes, e.g., when declaring a variable inside a for-loop, it will not be accessible outside of it. Similarly, recursive functions are supported, with DartBlock keeping track of each function call's memory space separately, while properly passing values between different scopes when passing from one function call to another.

3.6 Isolated Execution

The Dart language is single-threaded, with the UI created through the Flutter framework being rendered on the default main thread. However, DartBlock executes the user's program on a separate isolate as the user can potentially compose faulty programs: such cases may include the severe instance of an infinite loop, e.g., caused by a recursive function which is missing a valid ending condition. In that case, DartBlock's main thread for rendering the UI will not freeze indefinitely and cause the overall application to timeout or crash due to default operating system guardrails, but the erroneous program's execution is instead kept isolated to a separate thread with its own memory space.

An arbitrary timeout of 5 seconds has been programmed into DartBlock's execution: this design enables DartBlock to eventually stop the execution isolate, for example in the case of an infinite loop. In such cases, DartBlock also simulates an exception being thrown, explaining to the user that the execution had timed out.

3.7 Exceptions

Scratch [14]'s approach to executing a faulty program is to cause a soft failure, i.e., the execution is not disrupted and the statement causing the error is only highlighted with a red border, but a concrete error message is not shown. While Scratch's design in this regard is appropriate due to its younger target audience for whom it needs to mask certain technical details, DartBlock targets a more mature audience hoping to learn a concrete programming language, e.g., Java, in the context of a university course.

Instead, DartBlock replicates the experience of exceptions being thrown in an IDE when running text-based code: in addition to the program's execution being suspended upon an exception being thrown, the statement causing it is highlighted and the user is shown an exception description. The idea is to teach the user the exceptions that they would typically face in the case where they had been writing text-based code, e.g., Java.

Exceptions which DartBlock can throw include among others type mismatch exceptions, e.g., when the user tries to assign the wrong type of value to a given variable, as well as function-specific exceptions, such as when the user's custom function has a non-void return type but it does not return a value when called.

Furthermore, DartBlock can also throw a stack overflow exception, as shown in Figure 3, which it catches through the Dart runtime itself when executing the user's program, as well as a timeout exception in case the execution has not ended after the maximum duration of 5 seconds, as previously detailed in Subsection 3.6.

Thus, the idea is to more directly teach the DartBlock user concrete Java exceptions alongside a disruptive execution behavior when running faulty code, such that they can learn manual debugging by adjusting their program and re-executing it until the behavior is as expected.

3.8 Output

While Scratch [14] relies on a 2D sprite visualization and Alice [5] on a 3D rendering of objects, DartBlock opts for a console-based

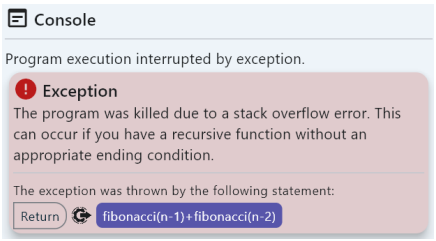


Figure 3: Example of an exception displayed in the console after execution, including the relevant statement.

text output, which is similar to how a traditional Java program's execution output is displayed in IDEs: "Print" statements in the user's program result in individual lines of text in the console and after the execution has finished, the console prints one final line indicating that the program's execution finished successfully. Also, similar to how stack traces for thrown exceptions are shown in an IDE's integrated console, DartBlock also displays the description of a thrown exception within the console, without hiding any previous "Print" statement outputs.

The console is automatically opened as an overlay upon execution, though the user can also manually open the console at any moment to review the last execution's output.

3.9 Export

Similar to other block-based frameworks such as Blockly [18], DartBlock also supports exporting the user's program to a traditional programming language. For our initial prototype, we provide Java as an option as it is the language taught in the class for our case study, though the introduction of other output options such as Swift and Dart is feasible.

3.10 Automatic Evaluation

DartBlock supports automatic evaluation of a program given a sample solution, with different evaluation schemas which can be used together to enforce a stricter evaluation. These include:

- Function Definition: it evaluates the definition of custom functions by checking their signatures (function name, return type, parameters).
- Function Output: it evaluates custom functions by checking their signatures, as well as their output. This schema requires the user, e.g., the teacher setting up the DartBlock quiz, to provide 1 or more sample function calls.
- Script: it evaluates the syntactical similarity of the input program to the sample solution by using their script output strings, e.g., Java code. The Damerau-Levenshtein [6] metric is used for this purpose, whose threshold (similarity percentage) can be customized.
- Print: it compares the console output of the input program to that of the sample solution by ensuring each line matches, including the order. It also deploys the Damerau-Levenshtein metric for computing similarities.
- Environment: it checks whether the same variables have been declared when executing the input program and the sample solution, based on their name, type and value.

The "Function Definition" and "Script" schemas are static evaluators, i.e., the user's program is not executed for the evaluation.

Multiple schemas were developed such that their combinations enable different types of questions for teachers, who can rely on these evaluators to enforce the strictness they want for each question. For example, the teacher may want to create an initial DartBlock exercise which is easy, e.g., where the user has to define a function without actually implementing its behavior. In that case, the "Function Definition" schema would suffice. On the other hand, for a harder exercise where the user is asked to implement a custom function with a specific behavior and to have their program perform certain actions in its main (entry) function, a combination of the "Function Output", "Environment", and "Print" schemas would cover the evaluation of the students' programs.

3.11 Difficulty Adjustment

DartBlock provides tools to randomly manipulate a given program, which are intended to be used by the quiz app [1] for adjusting the difficulty of DartBlock-based questions:

A sample program can thus be shuffled, i.e., the statements in its functions are randomly re-ordered, and also trimmed, e.g., given a percentage value of 50%, only the first half of the statements in each function is retained. Additionally, entire functions can be trimmed by also excluding their declaration (signature) and not just their body of statements, which would be used for the hardest difficulty where the user has to compose the solution from scratch.

Furthermore, DartBlock can automatically compose simple hints for the user, which for example tell the user the different types of statements expected in their solution as well as the number of variables expected to be declared. These hints are designed to give the user some starting point, while still being vague enough to not leak too much information about the solution.

The quiz app uses these different levers to represent discrete difficulties for DartBlock-based quizzes: for example, the easiest difficulty relies on keeping the entire sample solution, shuffling the order of statements and providing hints to the user, while a higher difficulty would also trim the sample solution by 25%. The hardest difficulty would be a blank starting program with no hints.

4 CASE STUDY

As a trial of DartBlock's first prototype, the library was integrated into a mobile quiz app [1] for the winter semester of 2023 at a university. Weekly quizzes were offered through the app to students of a 1st semester course in programming as part of the Bachelor in Computer Science, along with a final quiz at the end of the semester. The purpose of the quiz app was to offer students an additional means of practice for each lecture's topics and to improve their retention. The course itself is an introduction to basic programming through the Java language.

Usage of the app was both optional and anonymous, though a bonus grade could be earned by regularly playing the quizzes during the semester, which was offered to improve the participation rate. Content-wise, most of the quizzes consisted of multiple-choice and free-form questions, though with the inclusion of the DartBlock library, several new block-based coding questions were added to enable more interactive tasks.

Though the study is on-going at the time of writing, we can present the results for the DartBlock-related survey questions which the students were asked after playing the final quiz of the semester.

The following questions were rating-based ($1-5$), with the presented values and standard deviations (SD) rounded for readability. 16 students answered the survey:

(1) "How intuitive was the interface for building expressions (boolean, numeric) in the block-based code editor (DartBlock)?": 2.31 ($SD = 1.14$)
(2) "How useful was the option to convert your block-based code to real Java code?": 3.00 ($SD = 1.51$)
(3) "How useful was the option to execute your block-based code?": 2.75 ($SD = 1.24$)
(4) "How useful was the option to view the exceptions thrown by your faulty block-based code?": 2.56 ($SD = 1.21$)
(5) "How satisfied were you with the automatic evaluation of your block-based code?": 2.38 ($SD = 0.81$)
(6) "Overall, how easy was it to compose programs through the block-based code editor?: 2.00 ($SD = 0.97$)

The results generally indicate a neutral to slightly negative stance regarding the various aspects of DartBlock. For example, the first question's rating potentially alludes to the bespoke editors for composing numeric and boolean expressions, as described in Subsection 3.3, not appearing as easy to use as was hoped.

The scores hovering around the 2.50 middle-point do not necessarily indicate that users had significant qualms with DartBlock, though they were also not greatly impressed by the design to warrant high ratings.

5 DISCUSSION

The initial version of DartBlock satisfies the requirements for its integration in a quiz app, specifically through its automatic evaluation and difficulty adjustment options, though generally speaking it is also feature complete: it supports the basic building blocks (variables, decision structures, loops, functions) for starter programs and provides a runtime with exception throwing.

Nevertheless, the case study revealed a less than satisfactory reception of DartBlock based on the survey results. To some extent, this is to be expected as DartBlock required updates during the semester to address bugs which may have disrupted the experience.

On the other hand, the main reason may have been the organization of the experiment: DartBlock was integrated into the quiz app by introducing block-based coding questions in a small number of the weekly quizzes. This resulted in the users' initial exposure to DartBlock being abrupt as they did not get any tutorials beforehand, but they were thrown into DartBlock's UI and were expected to quickly learn how to use it to answer time-limited questions.

This problem was also exacerbated by the fact that some students had no prior programming experience, including other block-based coding frameworks to be familiar with visual programming. A better approach would be to provide short tutorials explaining the features of DartBlock before they are expected to use it through the quiz app, or to dedicate lecture time such that the students can experiment with DartBlock outside the context of time-limited quizzes to discover its functionalities without any prompts.

Such an introduction could familiarize them with DartBlock such that they would lose less time during the quizzes searching for the right blocks to compose their program. Additionally, they would make better make use of all the features, such as converting their program to Java code or executing their code multiple times to learn about exceptions and debugging. Hence, a consideration for future studies is to comprehensively introduce DartBlock, or any other similar framework, by offering documentation and open access such that the students can familiarize themselves with the interface before using it in the more restrictive context of timed quizzes.

Secondly, the UI of the boolean expression composer, shown in Figure 2, is too dense: boolean expressions can not only be composed of boolean constants, variables or function calls, but also of numeric comparisons, e.g., the greater than operator, and string equality checks. The problem lies in how the user can add numeric comparisons to their boolean expressions, namely via an extra button ("123") which opens up the numeric expression composer as an overlay. This can lead to several overlays as the user tries to create more complex boolean expressions consisting of multiple data types. The future version of the boolean composer should avoid overlays and opt for a single-view approach, e.g., by using a tabbed view to segment its components by type.

Finally, our initial case study was not designed to impact the students' learning, as DartBlock questions did not appear consistently in their quizzes during the semester. Rather, this study helped gather preliminary impressions for each of DartBlock's aspects to prepare targeted improvements for future studies. To improve the teaching of basic programming, DartBlock will not only have to be included in more quizzes during the semester, but also in non time-limited settings to enable users to also experiment with the framework as a free-form coding environment.

6 CONCLUSION

This work explored the new block-based programming library DartBlock, with a design and runtime more akin to the traditional programming experience: a typed variable system, an error-prone execution with exception throwing and a text input-free approach to composing expressions were chosen to adapt the learning goal to an older audience and to accommodate a mobile-first experience. Additionally, options to automatically evaluate programs against sample solutions, as well as to adjust their difficulty were built-in to support DartBlock's integration in quiz apps.

DartBlock was deployed in a university class to assess the initial impressions of students. The results revealed a less than ideal reception of DartBlock's various aspects, which we mainly attribute to organizational reasons: any new framework should be thoroughly presented to users before expecting its usage in a constrained context, e.g., within time-limited quizzes, such that students can discover the functionality at their own pace.

As for future work, we intend to further align DartBlock's visual programming style with the IDE experience, including step-wise execution, breakpoints and a way to view variable states as the program is running. We will also be conducting case studies where we provide more resources to the students to better learn about DartBlock and to potentially improve their perception of its features. An open source release of our framework is also planned.

REFERENCES

[1] Aryobarzan Atashpendar and Steffen Rothkugel. 2023. Improving Long-Term Retention through Personalized Recall Testing and Immediate Feedback. In *2023 11th International Conference on Information and Education Technology (ICIET)*. IEEE, 277–281.

[2] David Bau and David Anthony Bau. 2014. A preview of Pencil Code: A tool for developing mastery of programming. In *Proceedings of the 2nd Workshop on Programming for Mobile & Touch*. 21–24.

[3] Neil CC Brown, Michael Kolling, and Amjad Altadmri. 2015. Position paper: Lack of keyboard support cripples block-based programming. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 59–61.

[4] Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of computing sciences in colleges* 15, 5 (2000), 107–116.

[5] Stephen Cooper, Wanda Dann, and Randy Pausch. 2003. Teaching objects-first in introductory computer science. *Acm Sigcse Bulletin* 35, 1 (2003), 191–195.

[6] Fred J Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (1964), 171–176.

[7] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. 2012. Mediated transfer: Alice 3 to java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 141–146.

[8] Wenbo Deng, Zhongling Pi, Weina Lei, Qingguo Zhou, and Wenlan Zhang. 2020. Pencil Code improves learners' computational thinking and computer learning attitude. *Computer applications in engineering education* 28, 1 (2020), 90–104.

[9] Veronique Donzeau-Gouge, Gerard Huet, Bernard Lang, and Gilles Kahn. 1980. *Programming environments based on structured editors: The MENTOR experience*. Ph. D. Dissertation. Inria.

[10] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE blocks and beyond workshop (blocks and beyond)*. IEEE, 49–50.

[11] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer science education* 25, 2 (2015), 199–237.

[12] Charlotte Hill, Hilary A Dwyer, Tim Martinez, Danielle Harlow, and Diana Franklin. 2015. Floors and Flexibility: Designing a programming environment for 4th-6th grade classrooms. In *Proceedings of the 46th ACM technical Symposium on computer science education*. 546–551.

[13] Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. 29–38.

[14] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[15] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2010. Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research*. 69–76.

[16] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 168–172.

[17] Barbara Moskal, Deborah Lurie, and Stephen Cooper. 2004. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. 75–79.

[18] Erik Pasternak, Rachel Fenichel, and Andrew N Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE blocks and beyond workshop (B&B)*. IEEE, 21–24.

[19] Thomas W Price and Tiffany Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual international conference on international computing education research*. 91–99.

[20] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[21] Michael Tempel. 2013. Blocks programming. *CsTA Voice* 9, 1 (2013), 3–4.

[22] Nikolai Tillmann, Michal Moskal, Jonathan De Halleux, and Manuel Fahndrich. 2011. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. 49–60.

[23] David Weintrop and Uri Wilensky. 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*. 199–208.