Integrating Questions About Learners' Code in an Automated Assessment System

Afonso B. Caniço
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Portugal

André L. Santos
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Portugal

Abstract

Questions about Learners' Code (QLCs) assess programming students' program comprehension skills by providing personalised questions targeting the students' own program code. We conducted a preliminary, experimental implementation of integrating QLCs in the Automated Assessment System (AAS) used in an introductory programming course using Java. QLCs targeted some of the code assignments which students had to complete during the course. We collected 889 answers to QLCs, answered by 13 students over five course modules. We found that as the complexity of exercises increases, the success rate of the same type of QLC may not improve, and even exhibit a decline over time. We further analysed incorrect answers individually to relate them to possible misconceptions.

2012 ACM Subject Classification Social and professional topics → Software engineering education; Social and professional topics → Student assessment

Keywords and phrases programming education, student assessment, program comprehension, questions about learners' code

Digital Object Identifier 10.4230/OASIcs.ICPEC.2025.5

Funding This work was partially supported by *Fundação para a Ciência e a Tecnologia*, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020].

1 Introduction

Introductory programming students are often tasked with completing assignments which require the development of code conforming to functional requirements, such as the production of specific outputs. These types of assignments foster students' program writing skills, but do not necessarily increase students' confidence regarding their knowledge of the course materials [6, 5]. Further, students struggle when asked to explain their own program code [8], indicating that assignment completion does not necessarily imply proper knowledge acquisition.

Programming students tend to benefit from a more efficient learning process if they seek to understand their assignments' underlying concepts [20], which is not guaranteed by assignment completion. Further, students may struggle to understand and adequately guide their learning process towards deeper comprehension skills [11]. This obstacle is further amplified by the emergence of Large Language Models (LLMs), which are capable of producing working solutions to most introductory programming exercises [4]. An experiment carried out by Prather et al. (2024) [16] demonstrated that LLMs, when used in an unconstrained manner by introductory programming students, might be detrimental to students with pre-existing difficulties, as they create an illusion of understanding that can hinder the learning process.

Questions about Learners' Code (QLCs) [9] were proposed as a means of assessing the program comprehension skills of novice programmers. QLCs consist of questions (and, usually, multiple-choice options for it) which are formulated against a student's own code. In this way, students are assessed through questions which specifically target their own code, benefiting

from a learning aid adapted to the context they are working on. Several contributions have been made, particularly in automating this approach for programming courses employing different languages (e.g., Java [18], JavaScript [7], Python [10]).

In this paper, we describe our experimental integration of QLCs in the Automated Assessment System (AAS) used at our institution for introductory programming courses in Java. We aim to explore what insight we can gain from integrating QLCs into students' usual learning process. In particular, we explore the following research questions:

1. **RQ1:** How does students' success on different types of QLCs evolve over time?
2. **RQ2:** Can QLCs offer insight into novice programmers' misconceptions?

Our overall goal is to assess whether QLCs are an adequate complement to existing AASs, as we hypothesise that they can yield benefits such as: (a) providing students with personalised formative feedback, reinforcing program comprehension knowledge and offering a way to self-assess and detect misconceptions; and (b) provide instructors with a means to assess students' progress and detect learner misconceptions, possibly in an automated manner in a large-scale course.

We describe our preliminary study, in which we implemented QLCs in an AASs and collected data on student code submissions and QLC responses. We find that analysing students' answers to QLCs can offer insight into their learning progress and possible learner misconceptions, and how the latter evolve over time. These findings could provide an opportunity not only for students to regulate their learning process, but also for instructors to steer lectures towards addressing the most common or persistent student difficulties or misconceptions.

2 Related Work

Introductory programming courses traditionally focus on a writing-first approach, where students learn to program by writing code that conforms to specific requirements (usually functional). Previous studies have shown that successfully writing programs does not necessarily imply deep program comprehension [5, 14], and that learners struggle to explain their own code even when they were successful in developing programs [8]. Comparatively, comprehension-first teaching approaches, where program writing is preceded by the development of program reading skills, have been shown to be effective in developing program comprehension skills [19].

A recent approach in strengthening learners' program comprehension skills consists of Questions about Learners' Code (QLCs) [9]. Lehtinen et al. (2021) define QLCs as questions, posed automatically by a computer to a student, which refer to concrete constructs and patterns in the program code the student has written [8]. QLCs are typically multiple or single-choice questions, which can be verified automatically, or open-ended questions, which might require manual evaluation. They may be generated by employing a mixture of static analysis (e.g., which constructs are used) and dynamic analysis (e.g., how the code behaves when executed with specific inputs or starting states). Static QLCs focus on identifying how programming constructs are employed in the code, whereas dynamic QLCs focus on program behavior (e.g., tracing variable values). The ability to trace programs is highly correlated with program comprehension and writing skills [2, 12, 13]. A recent application of QLCs consists in combining them with the concept of notional machines [1], where the type of generated questions might involve, for example, the construction of a tree for a given expression written by the student.

```java
static int[] inverted(int[] v) {
    int[] iv = new int[v.length];
    int i = 0;
    while(i < v.length) {
        iv[i] = v[v.length - 1 - i];
        i = i + 1;
    }
    return iv;
}
```

(a) Student submission (function to produce an inverted array).

How many array positions are read during the function call **`inverted([4, 5, 6, 7])`**?

☐ 2 ✗
☐ 3 ✗
☒ 4 ✓
☐ 5 ✗

(b) QLC targeting the submitted code shown in Figure 1a.

Figure 1 Example of a QLC shown after a participant's submission to an assignment.

A previous study by Lehtinen et al. (2023) [8] investigated whether students' success in QLCs correlated to their program writing skills (in JavaScript) and their tendency to drop out. Their findings indicate that students show more difficulty in answering dynamic QLCs than static QLCs. Yet, overall, QLC success is highly correlated with course success and program writing skills. Further, they found that students with higher difficulty in answering QLCs throughout the course are more likely to drop out.

In previous work, we developed Jask (2022) [18], a first experimental system for automatically generating QLCs for introductory Java exercises. Similarly to other studies, we found that students struggle with dynamic QLCs more than they do with static QLCs. Figure 1 presents an example of a dynamic QLC. In this paper, we continue this line of work, developing more QLC types, and focusing on how to integrate the questions in an AAS.

3 Approach

Figure 2 presents the concepts of a typical AAS realisation (grey-shaded area). A *course* is composed of several code *assignments*. Each assignment has several *test cases*, which associate *arguments* to *expected* results, and define one (or possibly more) reference *solution*. *Students* are enrolled in the course and perform *submissions* to each *assignment*. Each submission holds the student solution, which is marked with a *score* (most likely, automatically).

The remaining concepts of Figure 2 relate to our approach. Each assignment may be associated to one or more *QLC Type* (from a *catalog*), meaning that the AAS will generate concrete *questions* of those types for submissions to that assignment. Each question has two or more multiple-choice *options*. The text of the question and the options result from instantiating templates against the submitted code, using its structure, identifiers, and execution data in the formulation of questions and their multiple-choice options.

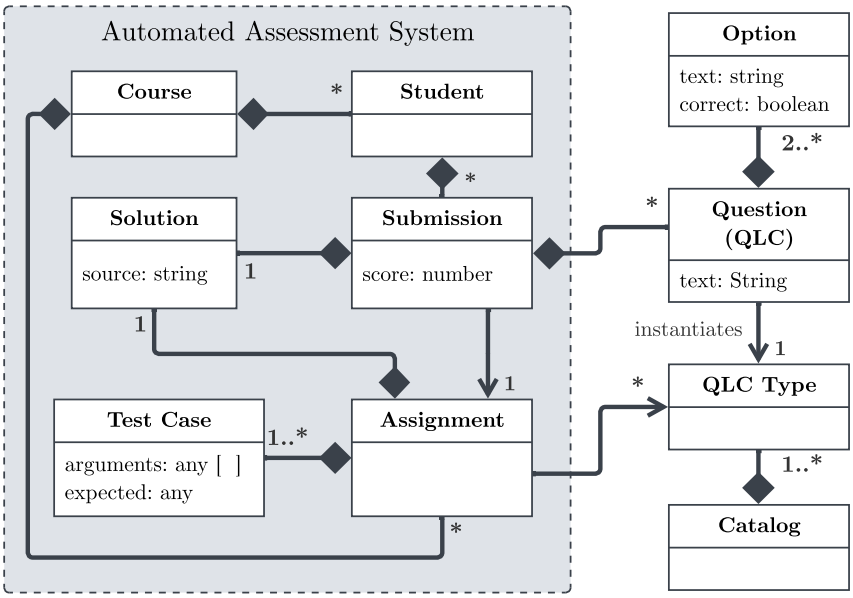

Figure 2 Model relating the concepts of a generic Automated Assessment System with those of our approach for QLC integration.

Since each QLC is generated from a student's submission, there is an extra layer of variability regarding the particular identifiers, structure, and behaviour of the student's code, which can influence both the question's formulation and options. Further, randomly selecting inputs from an assignment's test cases provides additional variability. These test cases are defined with meaningful values that cover usual and edge cases, and sampling from them guarantees that questions are posed with inputs which make sense for each assignment.

We implemented an experimental incorporation of QLC in the Paddle environment, an Automated Assessment System (AAS) in which students develop and submit solutions to coding assignments. Figure 3 illustrates the environment, where we can see a QLC being presented after an exercise submission.

After each submission that passes the functional tests (i.e. expected outputs are correct), students are prompted with QLCs targeting their solutions. The idea is that instructors associate QLCs that target concepts that are relevant for each specific assignment, taking the sequence of assignments into account. As this was our first experiment of using QLCs in a course, we chose to associate at most one QLC to each course assignment, and we did not allow multiple attempts in answering QLCs. All QLCs were single-choice (one correct option), with response being optional (student may leave the QLC blank). The answers were stored in a database and were used as data for our study.

4 Study

We conducted a study with introductory programming students by integrating our approach in a programming course for graduate students in non-STEM fields. The course was taught using Java, where students were tasked with assignments targeting the concepts of functions, expressions, variables, control structures, arrays, references, and procedures (side-effects).

Figure 3 Paddle environment displaying an correct submission with an incorrect QLC.

4.1 Context and Methods

In total, 13 students were enrolled in the course. Regarding gender, 7 (54%) of participants identified as male, with the remaining 6 (46%) identifying as female. The participants' ages ranged from 21 to 49. While the students had varying backgrounds, all came from non-STEM fields, with only 3 students having previously taken an introductory programming course.

The course lasted for one month, during which students had to submit a series of assignments in the Paddle environment. The course contents were divided into modules, each targeting different introductory programming concepts (see Table 1). Each of these modules had sets of mandatory and optional assignments and were completed in sequence. Students could not progress to the next module if the mandatory assignments of the current module were not accomplished. Since our aim was to use QLCs to assess students' comprehension of their programs and analyse student misconceptions, we chose to present QLCs only after a successful submission (i.e., it passes the test cases, and hence, is functionally correct). The set of QLCs used in our study is available as a standalone library[1].

[1] `https://github.com/ambco-iscte/jask`

Table 1 Course modules.

Module	Content
M1	Functions and Expressions
M2	Variables and Control Structures
M3	Dependent Functions
M4	Arrays
M5	References and Procedures

4.2 Results

In total, we collected a total of 889 QLC answers, with an overall success rate of $\approx 75\%$. Table 2 presents each QLC type included in our study, along with the corresponding success rate across the duration of the course. For the QLCs which appeared in more than one course module, we show their success rate over each module (% of right answers per module) in Figure 4.

Given the small number of students and the relatively small number of QLC attempts, we manually investigated each incorrect (not blank) response to identify possible student misconceptions which could have led to those answers. A summary of the identified misconceptions is given in Table 3. These misconceptions were identified by analysing the students' incorrect QLC responses. The two authors of this paper served as raters of these incorrect attempts, attributing one or more of these misconceptions. The classification yields a percent agreement of 68.9% and a Cohen's kappa value of $\kappa = 0.58$, with a 95% confidence interval of $[0.48, 0.67]$, indicating moderate to substantial agreement [15].

We further analyse the incidence over time of each identified misconception that occurred in more than one course module, which we define as the ratio between the number of incorrect QLC attempts in each module which *could* be attributed to a given misconception, and the number which effectively were attributed to that misconception. That is, for a course module M and a misconception m, we calculate

$$\text{Incidence}(m, M) = \frac{\#\text{ of incorrect QLC answers in } M \text{ attributed to } m}{\#\text{ of QLCs in } M \text{ in which } m \text{ could appear}} \in [0, 1].$$

Since each author may attribute different labels to each incorrect QLC answer, the incidence depends on which classification is used. Thus, we choose to present the "true" incidence value as the mean the incidence values given by both raters. We present these results in Figure 5.

4.3 Threats to Validity

The main limitation of our study is the number of participants, since any results drawn from a sample of $n = 13$ students are difficult to generalise. Further, it is difficult to draw conclusions regarding possible correlations between QLC and overall course success or other indicators of student performance, since analyses of statistical significance are inadequate for such a small sample. Nevertheless, we do not consider that this imposes any significant limitation to the results we have observed.

Regarding our approach, we could collect a larger pool of QLC attempts by assigning more than one QLC per assignment. However, we argue that, for our first attempt at integrating QLCs within an AAS, a smaller, more controlled experiment would be more easy to managed.

Table 2 QLCs used in the course and their overall success rates.

QLC Type	Template	Total	Correct
WhatIsResult	What is the value returned by the function call `f(args)`?	48	93.8%
HowManyParams	How many parameters does the function `f` have?	57	89.5%
HowManyVariables	How many variables (not including parameters) does the function `f` have?	43	88.4%
WhichReturnType	What is the return type of the function `f`?	73	87.7%
WhichVariableHoldsReturn	Which variable holds the result of the function `f`?	39	87.2%
WhichParameters	Which are the parameter names of the function `f`?	75	85.3%
WhatVariables	Which are the variable names (not including parameters) of the function `f`?	39	79.5%
WhichParameterTypes	Which are the parameter types of the function `f`?	65	78.5%
WhichFunctionDependencies	Which other functions does the function `f` depend on?	28	75.0%
HowManyArrayWrites	How many array position writes are performed when calling `f(args)`?	36	75.0%
HowManyFunctionDependencies	How many functions does the function `f` depend on?	26	69.2%
WhichVariableValues	Which is the sequence of values taken by the variable `v` when calling `f(args)`?	69	66.7%
HowManyArrayReads	How many array position reads are performed when calling `f(args)`?	58	65.5%
HowManyVariableAssignments	How many times is the variable `v` assigned when calling `f(args)`?	75	64.0%
HowManyLoopIterations	When calling `f(args)`, how many iterations does the loop perform?	95	58.9%
HowManyFunctionCalls	How many calls to f_2 are performed when calling f_1`(args)`?	47	55.3%
HowManyArrayAllocations	How many arrays are allocated when calling `f(args)`?	16	37.5%

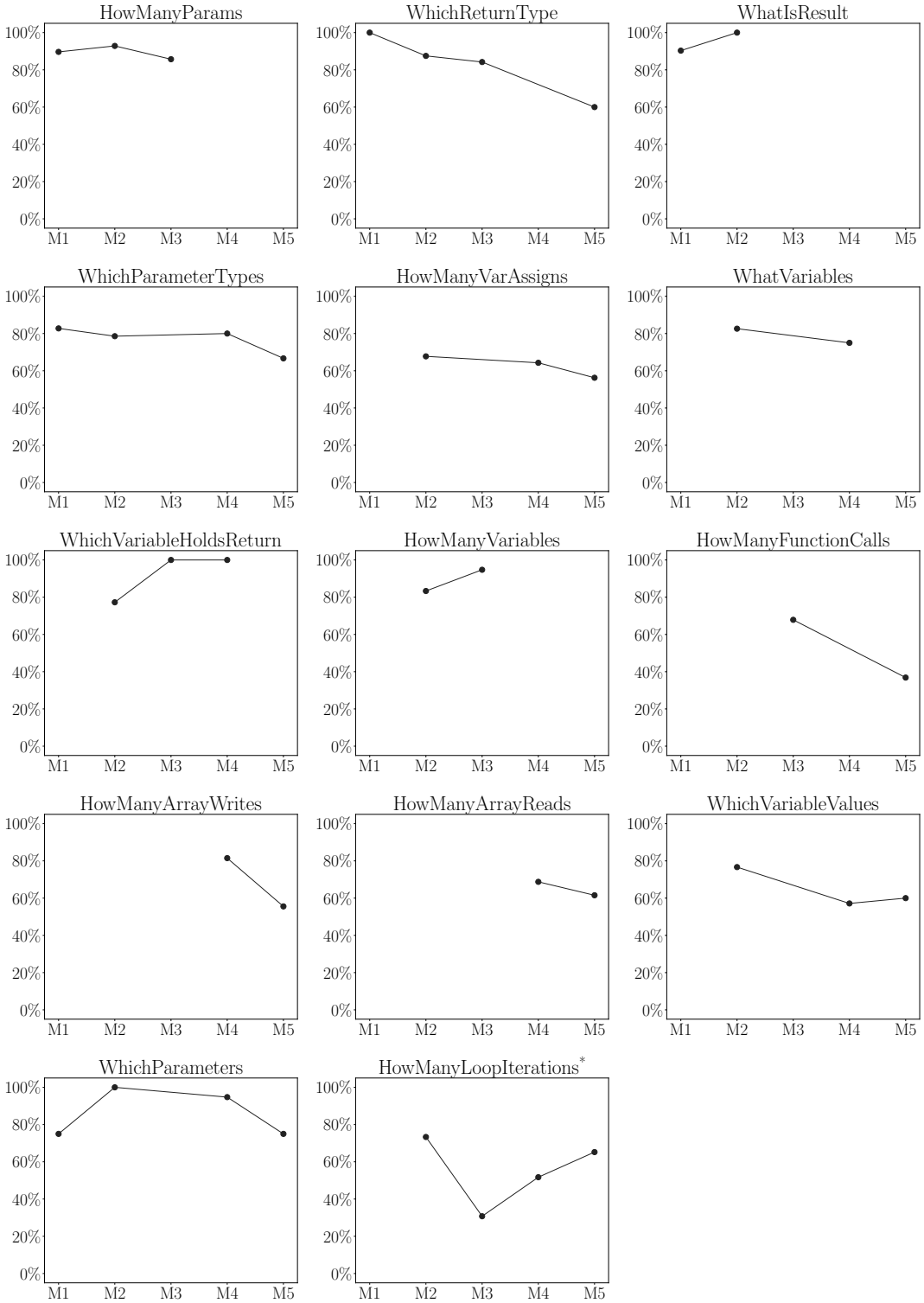

Figure 4 Evolution of the success rate of each QLC type over the course modules. Only QLCs which appeared in more than one module are shown.

Table 3 Possible student misconceptions inferred from incorrect QLC answers.

Misconception / Mistake	The student...
NotCountInitialisationAsAssignment	Fails to consider the initialisation of a variable as an actual assignment.
OneMoreLoopIteration	Is "Off-By-One": counts one more iteration than in reality.
ConfuseParameterNamesWithTypes	Fails to differentiate between a method's parameters' names and types.
OneLessLoopIteration	Is "Off-By-One": counts one less iteration than in reality.
NotCountFinalLoopIteratorAssignment	Fails to consider the (last) assignment of a loop's iterator variable when that assignment will cause the guard evaluation to return False.
ConsiderArrayReadAnAssignment	Erroneously considers a read access to an array to be an assignment to the corresponding array variable.
ConsiderMethodItsOwnDependency	Erroneously considers a method to be dependent on itself, even when it's not recursive.
ConsiderNativeInstructionsAsFunctions	Erroneously considers native instructions, such as `return`, to be functions.
NotCountUnaryOperatorAsAssignment	Fails to consider expressions such as `i++` or `i--` as assignments to the variable `i`.

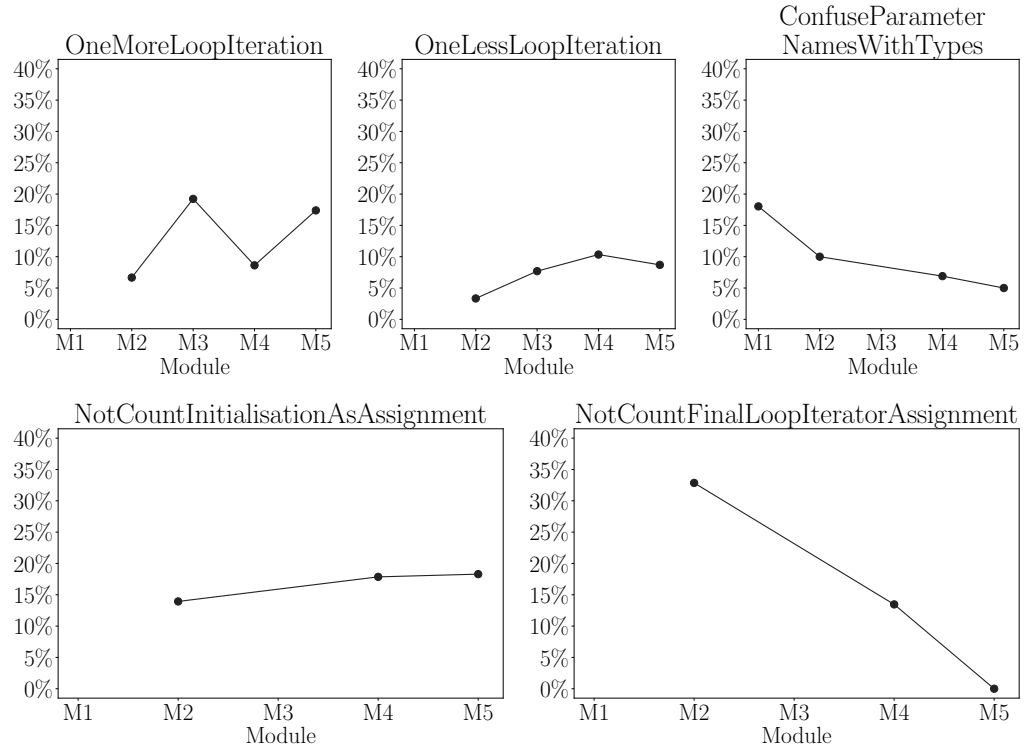

Figure 5 Incidence rate of student misconceptions per course module (average values of the two raters). Only misconceptions which occurred more than once are pictured.

In informal observations, we noticed that students did not appreciate having a single opportunity to answer each QLC. Most students have a clear wish of keeping a "clean sheet" regarding module completion, where they have all exercises and questions marked as successfully completed. The single chance of answering a QLC led some of them to refrain from answering them without checking with the instructor first. This may have slightly influenced the success rate of QLCs towards more correct answers.

Regarding student misconceptions, care should be taken in the classification of incorrect QLC attempts. Students can make simple mistakes or distractions, and thus it might not be valid to necessarily attribute every incorrect attempt to a misconception. Both the percent agreement and Cohen's kappa measure indicate moderate to substantial agreement when it comes to attributing possible misconceptions to incorrect QLC answers. Nevertheless, more reliable results might have been obtained by: (1) employing more raters; (2) refining the classification process; (3) having a larger sample. Further, the reliability of the classification could be strengthened by having students write a short explanation/reason for their answers.

5 Discussion

Our results show that integrating QLCs within the AAS of a programming course can offer insight into how learners' knowledge and misconceptions evolve over time.

Regarding the overall QLC success rates shown in Table 2, we note that it is not straightforward to attribute specific causes to differing success rates, as they depend on the assignment's context and its appearance within the course. Nevertheless, unsurprisingly, these values showcase students generally struggle more with questions related to concepts like tracing (e.g., `WhichVariableValues`) as opposed to static program properties (e.g., `HowManyParams`). This confirms the observations of previous studies [18, 7].

5.1 Student Success over Time

Regarding **RQ1**, it might be expected that students' knowledge regarding different programming concepts increases, or at least does not diminish, over a course's duration. However, the evolution of QLC success rates shown in Figure 4 contradicts this belief, showing student progress can fluctuate significantly or even decrease over time. In particular, the evolution shown in Figure 4 for QLCs of type `HowManyLoopIterations` highlights particularly unpredictable progress, with students' success rate diminishing abruptly after seemingly achieving higher values ($\approx 75\%$ in M2 to $\approx 30\%$ in M3). We investigated this inconsistency in student performance by performing a manual analysis of the incorrect responses for this QLC type within modules M2 and M3. The main difference is that loop structures used in the assignments of module M2 involve less variables and instructions than those in module M3, in which several variables may be updated and conditions checked within a loop's body. Broadly, we may attribute the decrease in performance to the higher complexity of the code produced for M3's assignments.

As an illustrative example, Figure 6 shows the possible difference in complexity between assignments of different modules, with two submissions of the same student. Both assignments included a `HowManyLoopIterations` QLC upon completion. The student correctly answered the QLC for the first exercise (Figure 6a), but not for the second (Figure 6b). We note that this may not necessarily correspond to a misconception, as it might be the case that the increased code complexity might lead the student to "slip" without evidencing a misunderstanding.

```
int i = 1;
while(i < 5) {
   i = i + 1;
}
```

(a) Student's submission for an exercise in module M2 – Variables and Control Structures.

```
static boolean isPrime(int n) {
    int c = 0;
    int i = 1;
    while (i <= n) {
        if (n % i == 0) {
            c = c + 1;
        }
        i = i + 1;
    }
    if (c==2){
        return true;
    }
    return false;
}
```

(b) Student's submission for an exercise in module M3 – Dependent Functions.

Figure 6 Two submissions of a student to an assignment in module M2 and another in M3, respectively. Both exercises included a QLC of type `HowManyLoopIterations` upon completion.

Notice how apparently simple static QLCs, such as `WhichReturnType` and `WhichParameterTypes` decline in the rate of correct answers over the progression of modules. The introduction of more types (single and multi-dimensional arrays, references) as modules progress may lead to new student misunderstandings involving concepts that initially did not reveal any significant difficulty.

5.2 Student Misconceptions

Regarding **RQ2**, our results show that the analysis of QLC success within the context of each assignment can offer an overall insight into the most prominent novice programmer misconceptions and how they evolve over time.

In Figure 5, we see that students struggle with counting the exact number of iterations a loop executes, frequently counting one more than required (`OneMoreLoopIteration`). Further, approximately 1 in 5 (20%) QLC attempts present a failure to count a variable's initialisation as a value assignment (`NotCountInitialisationAsAssignment`), a misconception which was persistent over the course. Even misconceptions relating to simpler concepts, such as determining the names and/or types of a function's parameters (`ConfuseParameterNamesWithTypes`), while decreasing over time, still showed some incidence in later modules.

These results reveal that, while students are generally able to produce working code in their assignments, some misconceptions persist over the duration of the course. This agrees with previous findings that assignment completion does not imply full comprehension [8].

5.3 Limitations

The study was conducted with a small, yet balanced, sample consisting of 13 students. While our observations can provide valuable insight, it might be beneficial to collect more generalisable data by conducting a larger-scale study with introductory programming students.

The main limitation regarding QLC formulation is likely to be the quality of the provided distractors. Meaningful distractors should make sense for each question type and corresponding assignment, as well as relate to common student misconceptions [17, 3]. Purely random distractor generation would not be adequate, as the same values might not make sense for differing question types or assignments. For example, an "off-by-one" approach might only be adequate for iterator variables. Additionally, it is essential to introduce sufficient variability in the QLCs presented to each student, fostering academic integrity (lower chance of plagiarism) and increasing the amount of usable content (unlikely to generate the exact same QLC twice). This variability is achieved not only by generating each QLC from each student's code, but by enriching each assignment with a set of meaningful test cases which cover a variety of possible inputs.

6 Conclusions

Our observations show that QLCs can offer insight into how students' knowledge on elementary programming concepts evolves over time during an introductory programming course. We also identified and analysed common student misconceptions and how they persist over time. Our results show that student progress is not linear, with prior knowledge not necessarily persisting over time or transferring to assignments of higher complexity, and with misconceptions not necessarily improving over time. The sample used to derive the results was small, but we believe this introduces no significant bias or limitation to our analysis. Our findings thus reveal that, while students may appear to understand certain concepts within simple contexts, this knowledge may not necessarily transfer to assignments of higher complexity.

An implication for instructional design is that it is worth posing the same type of QLC several times over the progression modules, as concept acquisition tends to be fragile. By monitoring the QLC results, instructors may detect students' most prominent misconceptions, in order to prioritise and reinforce aspects in lectures that students struggle the most. On the other hand, students may use QLCs as a means of self-assessment that strengthens their skills and uncovers knowledge gaps. Further, we should provide multiple attempts for answering QLC, as is not beneficial to have any sort of attrition in the platform usage.

We intend to continue augmenting our AAS with QLCs that not only aid novice programmers at each step of their learning process, but provide instructors with insight on how to structure and guide their course in order to adequately tackle student obstacles. We envision the usage of QLCs to reinforce and offer insight into students' program comprehension skills, and to detect and mitigate misconceptions. As such, in future work we are likely to include the implementation of QLCs targeting other aspects of novice programmers' learning process, like error message comprehension and code quality towards refactoring. We also plan to enrich QLCs with comprehensive explanations for the correct answers, as well as possible misconceptions for the incorrect options.

References

1 Joey Bevilacqua, Luca Chiodini, Igor Moreno Santos, and Matthias Hauswirth. Using notional machines to automatically assess students' comprehension of their own code. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2024, pages 1572–1573, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3626253.3635524.

2 Teresa Busjahn and Carsten Schulte. The use of code reading in teaching programming. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 3–11, New York, NY, USA, November 2013. Association for Computing Machinery. doi:10.1145/2526968.2526969.

3 Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. A curated inventory of programming language misconceptions. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 380–386, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3430665.3456343.

4 James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, pages 10–19, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3511861.3511863.

5 Cazembe Kennedy and Eileen T. Kraemer. Qualitative observations of student reasoning: Coding in the wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '19, pages 224–230, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3304221.3319751.

6 Päivi Kinnunen and Beth Simon. My program is ok – am i? computing freshmen's experiences of doing programming assignments. *Computer Science Education*, 22(1):1–28, 2012. doi:10.1080/08993408.2012.655091.

7 Teemu Lehtinen, Lassi Haaranen, and Juho Leinonen. Automated questionnaires about students' javascript programs: Towards gauging novice programming processes. In *Proceedings of the 25th Australasian Computing Education Conference*, ACE '23, pages 49–58, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576123.3576129.

8 Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. Students struggle to explain their own program code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, pages 206–212, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3430665.3456322.

9 Teemu Lehtinen, André L. Santos, and Juha Sorva. Let's ask students about their programs, automatically. In *Proceedings - 2021 IEEE/ACM 29th International Conference on Program Comprehension, ICPC 2021*, Proceedings/IEEE International Conference on Program Comprehension, pages 467–475, United States, May 2021. IEEE. International Conference on Program Comprehension, ICPC ; Conference date: 20-05-2021 Through 21-05-2021. doi:10.1109/ICPC52881.2021.00054.

10 Teemu Lehtinen, Otto Seppälä, and Ari Korhonen. Automated questions about learners' own code help to detect fragile prerequisite knowledge. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, pages 505–511, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3587102.3588787.

11 Rachel S. Lim, Sophia Krause-Levy, Ismael Villegas Molina, and Leo Porter. Student expectations of tutors in computing courses. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, pages 437–443, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3545945.3569766.

12 Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *SIGCSE Bull.*, 41(3):161–165, July 2009. doi:10.1145/1595496.1562930.

13 Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth international Workshop on Computing Education Research*, ICER '08, pages 101–112, New York, NY, USA, September 2008. Association for Computing Machinery. doi:10.1145/1404520.1404531.

14 Sandra Madison and James Gifford. Modular programming. *Journal of Research on Technology in Education*, 34(3):217–229, 2002. doi:10.1080/15391523.2002.10782346.

15 Marry L. McHugh. Interrater reliability: the kappa statistic. *Biochemia Medica*, pages 276–282, 2012. doi:10.11613/bm.2012.031.

16 James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. The widening gap: The benefits and harms of generative ai for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*, ICER '24, pages 469–486, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3632620.3671116.

17 Yizhou Qian and James Lehman. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Educ.*, 18(1), October 2017. doi:10.1145/3077618.

18 André L. Santos, Tiago Soares, Nuno Garrido, and Teemu Lehtinen. Jask: Generation of questions about learners' code in java. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, ITiCSE '22, pages 117–123, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3502718.3524761.

19 Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, pages 1–49, January 2019.

20 Daniel Zingaro, Michelle Craig, Leo Porter, Brett A. Becker, Yingjun Cao, Phill Conrad, Diana Cukierman, Arto Hellas, Dastyni Loksa, and Neena Thota. Achievement goals in cs1: Replication and extension. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 687–692, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3159450.3159452.