An Open List of Computer Programming Student's Common Problems and its Leverage in Teaching Practice

Renato Cortinovis, Pablo Frank Bolton, Ricardo Caceffo
rmcortinovis@gmail.com; pfrank@smith.edu; caceffo@ic.unicamp.br
Independent Researcher; Smith College; Unicamp

ABSTRACT

Educators are frequently baffled by the problems faced and misconceptions held by their novice programming students. Yet, understanding these problems clearly is fundamental for an educator, both to properly evaluate their students as well as to engage them with suitable pedagogical strategies. This paper describes the development of a comprehensive – although not final – open list of problems commonly experienced by novices, focused on procedural and object-oriented programming, as much language agnostic and conceptually/strategically oriented as possible. It explores ways of using the list to improve the evaluation of students in the teaching practice, with targeted tests and detailed evaluation rubrics. The associated antipattern cards provide examples on how to detect a specific problem, indications on its possible origin, and suggestions for pedagogical strategies to overcome it. The paper finally hints at a strategy to crowdsource a battery of standardized/calibrated tests, that can be used both for the formative and summative evaluation of students, as well as to objectively compare different educational strategies and educational systems.

CCS CONCEPTS

• **Social and professional topics** → Computing education.

KEYWORDS

CS1, Programming Language, List, Problem, Misconception.

1 INTRODUCTION

Educators are frequently baffled by the problems faced and misconceptions held by their novice programming students. By the term "misconception", we mean a programming mistake at a conceptual level which is systematically repeated, implying a stable but incorrect understanding, while a programming mistake is an error in the code that leads to incorrect or unexpected behavior, because of syntactic, semantic, or logic problems [1]. In this paper we use the generic term "problem" as a catch-all term, including all sorts of misconceptions, problems and challenges faced by novice programmers, such as failing to properly trace an existing program.

The author(s) or third-parties are allowed to reproduce or distribute, in part or in whole, the material extracted from this work, in textual form, adapted or remixed, as well as the creation or production based on its content, for non-commercial purposes, since the proper credit is provided to the original creation, under CC BY-NC 4.0 License.
EduComp'23, Abril 24-29, 2023, Recife, Pernambuco, Brasil (On-line)
© 2023 Copyright held by the owner/author(s). Publication rights licensed to Brazilian Computing Society (SBC).

Understanding these problems clearly is important to be able to help students overcome them [2]. First, it is necessary to be able to diagnose them. Second, knowing their roots is instrumental in planning pedagogical activities suitable to correct them [1]. For example, pedagogical activities can be based on making misconceptions obvious [3], as students can learn from their mistakes when they understand the faulty mental models causing the errors [4]. Additionally, knowing precisely the problems faced by novices can allow for the design of learning activities explicitly based on their possible errors, as in the Productive Failure pedagogical strategy [25]. Third, understanding these problems may allow students to avoid them in the first place, before they become difficult to eradicate. For example, Holland et al. [5] state that when learning an Object-Oriented programming language, novice students should not be exposed to too many exercises where there is just one object instantiated from each class, to avoid developing the problem – mentioned by Sanders and Thomas [6] – of class and object conflation.

It is recognized in the literature that the most meaningful problems are not related to the specific mechanics of a programming language. Goldman et al. [7], for example, claim that students find the greatest difficulties in tracing and problem solving. Mccauley et al. [4] note that the major problems are related to understanding the task and basic design and are not specifically related to the programming language. Hanks [8] suggests that major challenges are rather design-oriented, due to lack of design knowledge/problem solving skills. Therefore, the goal is to focus on problems as language agnostic as possible, and more design oriented rather than lexically or syntactically oriented.

Thus, this work aims to answer the following research questions:

- **RQ1:** What are the main common problems faced by programming novices in the fundamental areas of procedural and object-oriented programming, regardless of the programming language?
- **RQ2:** How could a list of common problems be used to enhance teaching and learning?

This paper is organized as follows: Section 2 discusses related work; Section 3 presents the methodology used to identify the common problems and derive the proposed list; Section 4 describes the resulting list of common topics and problems identified. In Section 5 we discuss the answers to the research questions, including how the list can be used in the teaching practice. Finally, Section 6 outlines future activities and Section 7 presents our conclusions.

2 RELATED WORK

The literature was examined starting from recent secondary sources and tracing back to relevant primary sources with forward and backward citation search. We also examined papers on Concept Inventories, again secondary and primary sources, including our own publications.

Qian and Lehman [1] present a thorough analysis of the extended literature concerning programming errors, misconceptions and other problems related to novice programmers. They start by rigorously clarifying the meaning of terms such as errors, mistakes, bugs, misconceptions, difficulties, challenges, or misunderstandings, and adopt a useful classification framework in terms of syntactic (specific language features), conceptual ("how programming constructs and principles work and what happens inside the computer"), and strategic ("how to apply syntactic and conceptual knowledge of programming to solve novel problems": planning, writing, tracing, debugging) knowledge. They claim that identifying specific students' problems is a fundamental competence for computer science teachers, and that the overall goal should be to investigate the factors that contribute to these problems, and strategies to address them. They present examples of problems/difficulties in their final tables, but these are fairly limited in scope.

Another source of misconceptions is the literature about Concept Inventories (CI), test-based assessments of concepts, where the distractors (incorrect choices for a question) are indeed based on common student misconceptions. The misconceptions for CS1 by Tew and Guzdial [9] in particular, are interesting because they are language independent. Yet, they are not easily available to avoid "saturation", that is, to avoid becoming known among the students, as they would lose their reliability. To reduce the problem, Parker et al. [10] developed a replica validated against the previous one. There are other CIs on specific topics, for example recursion by Hamouda et al. [11], data structures by Porter et al. [12] and algorithm analysis by Farghally et al. [13].

Some authors identified programming misconceptions and designed language-specific CIs based on them. For example, Caceffo et al. identified misconceptions for Introductory Programming Courses (CS1) in C [14] and designed a CI in that language; Gama et al. [22] identified misconceptions in Python programming language, and Caceffo et al. [23] identified misconceptions in the Java language.

Yet, the goal of CIs is to efficiently detect a few meaningful misconceptions, without covering them exhaustively, hence they can be a good source of ideas but are usually pretty limited in coverage. Most importantly, they should not be used for formative or summative evaluation to avoid saturation. Additionally, they only tend to target the assessment of students' conceptual understanding, but not their problem solving or design skills [7].

There are also attempts to draw comprehensive lists of mistakes. Sanders and Thomas [6], in particular, provide useful checklists for grading OO CS1 programs, but they do not cover procedural programming. Pillay and Jugoo [15], on the contrary, only consider procedural programming, but in a specific language (Java). Yet, for example, they do not include recursion. Robins et al. [16] too, mainly focus on language related problems. Brown and Altadmri [24] focus on 18 novice Java programming mistakes, adapted from the 20 mistakes previously identified by Hristova *et al.* [18]. These mistakes have been thoroughly analysed exploiting a large dataset of Java compilation events (Blackbox of BlueJ), yet being based on compilation errors they are mainly syntax-oriented.

Another issue relevant to this paper, again widely discussed in the literature, is the taxonomy used to organize the problems. Most taxonomies are organized around lists of concepts/topics. Goldman et al. [7], in their effort to develop CIs for introductory computer courses, produce a list of topics which are both "challenging" – identified with novice-centric techniques – and "important" – identified with expert-centric DELPHY-based techniques. Yet their list is quite general and not focused on programming. Luxton-Reilly et al. [17] offer a list of concepts focused on programming, derived from nine literature sources with a rigorous process, which they use to design assessment tests precisely focused on single concepts (mastery learning). Table 1 summarizes some of the works described, showing how this study is situated.

Table 1: Related Work concerning programming errors and how this study is situated

Paper	Type	Language
[1]	Programming errors, misconceptions and other problems related to novice programmers.	Language independent
[9]	CS1 misconceptions	Language independent
[10]	CI replica validated against the previous one.	Language independent
[11]	CI on recursion	C, Java
[12]	CI on data structures	Pseudocode
[13]	CI on algorithm analysis	Java
[14]	CI for introductory programming courses	C
[22]	Misconceptions	Python
[23]	Misconceptions	Java
[6]	Checklists for grading OO CS1 programs	Java

[15]	Problems in procedural programming	Java
[24]	Mainly syntax-oriented mistakes	Java
This Work	Design oriented problems faced by novice programmers	Language independent – procedural and Object-Oriented paradigms

3 METHODOLOGY

The methodology adopted in this study has 3 steps, illustrated in Figure 1. In Step 1, the authors analyzed the literature (see Table 1) about misconceptions and students' problems, discussing the data and findings in the light of their own experience. One of the authors, for example, had analyzed and graded, over more than two decades, more than 30.000 students' solutions to proposed programming exercises, both on paper and on a computer, and discussed many of them individually with the students. In this step, the authors analyzed the misconceptions listed in C [14], Python [22] and Java [23] languages. These lists were compared, discussed, refined and merged, via remote asynchronous discussions, aiming to i) identify the commonalities among them and ii) generalize the misconceptions, i.e., reduce the language-dependent and syntax information from them, as far as possible.

In Step 2, the resulting set was considerably extended and harmonized with other existing lists in the literature, in particular those produced by Pillay and Jugoo [15], Sanders and Thomas [6], Robins et al. [16], and Luxton-Reilly et al. [17], especially concerning design-oriented aspects. In this step, the authors organized the problems identified in a general list of language independent topics, again identified from the literature and their own experience.

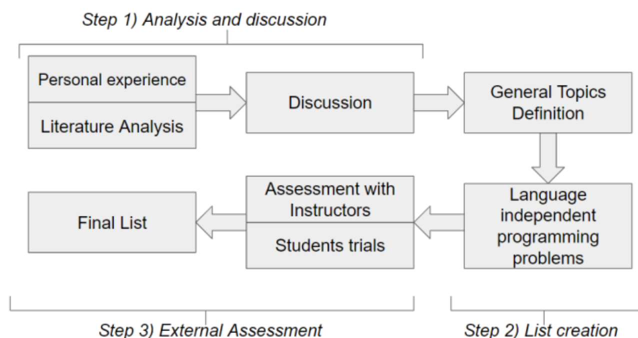

Figure 1: The methodology to derive the list.

In Step 3 the resulting list was first checked against actual mistakes resulting from the correction of programming tests in three courses with a total of 52 students in Italy. The list was finally assessed by 5 additional computer science instructors, 3 from Italy with more than 20 years' experience, and 2 from the USA, and the final version was produced.

4 RESULTS

With respect to the list provided by Pillay and Jugoo [15, Appendix A], many existing items, such as "Incorrect syntax of equations", "Incorrect syntax of Boolean equations, "Syntax errors such as missing semicolon", "Method returning a value declared as void", or "Syntax errors in the combination of variable and string output", were all merged in a single generic "Basic syntax errors", as in Hristova et al. [18] or Robins et al. [16], because the focus of our new list is more on conceptual and strategic knowledge, rather than syntactic knowledge [1].

As an additional example, "Condition added after else" was eliminated because it is syntax oriented, while "Redundant use of an IF statement, instead of an ELSE clause" was introduced because it is more logic oriented. Some items were merged and generalized, for example "Incorrect initial value assigned to an accumulator" and "Use of variables that have not been assigned values only declared" were merged and generalized in "Missing / incorrect variable initialization"; "Accumulator initialized in the loop" and "Calculations that should be performed after the loop are performed in the loop" were merged and generalized as "Insertion in the loop body, of code that should be executed only once before or after the loop". Other items were just generalized, for example the item "Separate if-statements used instead of using the OR operator", considering that in many cases an AND operator could be required, has been generalized to "Redundant structured if-statements where Boolean expressions could simplify the code". As another example, "Assigning more than one calculation to a variable" was generalized as "Overwriting the content of a variable before using it".

With respect to the list provided by Sanders and Thomas [6, Table 2], which deals exclusively with Object Oriented programming, items such as "Variables with names that are really values of attributes" had already been taken into account in the non-OO topic "Simple variables". The other items were included in the list, but extended with new ones, such as "Confusion concerning the identification of suitable parameters for instance versus class methods", or "Improper use of current (this / self / Me) object".

Similarly, other items were taken (sometimes modified) from the list provided by Robins et al. [16, Appendix]. For example, we incorporated the items "Wrong basic structural details", and "Stuck on program design (solution understood, but can't turn that understanding into a program)". The item "Problems with exceptions, throw catch" was further detailed in order to take into account the ability to trace existing code with exceptions, to use existing exceptions, or to develop customized ones. Others, too language specific, were generalized – for example the items "Hierarchies" or "Event driven programming" that mentioned Java specific mechanisms.

Brown and Altadmri [24] analysed a large dataset of mistakes identified by a Java compiler, hence strongly syntax-oriented. Indeed, for example, the most frequent error they identified is non-matching parenthesis, which is certainly a frequent mistake, yet scarcely symptomatic of meaningful misconceptions, and considered as a generic "Syntax Error" in our list. Other mistakes

they could identify through compilation errors are potentially more symptomatic of meaningful misconceptions, such as incorrectly ignoring a value returned by a method – these are therefore included in our list.

With respect to the list of misconceptions identified in C [14], Python [22], and Java [23] languages, the following item was included without modifications in the "Variables and Expressions" topic: "Attempt to access local variables from outside scope". Similarly, from [14] and [23], the item "Global variables considered local in current scope" was included without changes in the same topic. The items "Global variables assumed inaccessible from within function" [14, 23] and "Iteration variable used in for statement considered local" [22, 23] were generalized into the following item: "Failure to understand the scope-rules".

In turn, the items "Wrong order/precedence of operators in expressions (including, for example, misuse of parenthesis)", "Incorrect order of function parameters", "Attempt to access parameter from outside scope" and "Parameters passed as if by reference", all of them present in [14, 22, 23], were included without changes. The item "Parameter value set by external source" [14, 22, 23] (e.g., when in the first line of a function an input command overwrites the value of that parameter) was generalized and redefined to "Overwriting the value of a parameter before using it.", thus including other situations in which the problem could manifest. Then, the item "No self keyword to reference instance attributes" [22] was generalized to "Improper use of current (this/self/me) object". Finally, many items were introduced in the list ex-novo, for example: "Confusion between sequence versus nesting of IF-statements", "Code repeated in both the THEN and ELSE clauses", or "Inability to trace the execution of IF statements".

As mentioned in the methodology section, the list obtained was used to correct tests from 52 computer science students of three classes. Every student mistake could be classified with the existing list, and every item in the list corresponded to at least one actual mistake in a student test.

The list was finally critically revised by 5 educators, who were asked whether they could see any opportunity to include additional items, delete or merge existing items, or modify them. The three experienced computer science educators from Italy and the two from the United States who were asked to critically revise the list, did propose a few additional entries, who were integrated in most cases as examples of existing ones. One of the educators noticed that he could associate students' visages to each listed mistake. Another one showed appreciation for the list, because he could easily exploit it as an evaluation rubric for the self-evaluation of his students.

4.1 Resulting Final List

In total, 9 common topics of problems faced by programming novices were identified in the resulting final list: background problems, variables and expression, data structures, input and output, control structures, modularization, object-oriented fundamentals (classes and objects), object-oriented design, and problem-solving. Considering all topics, 107 programming problems were identified, as shown in Table 2:

Table 2: Final list of Topics (N=9) and Programming Problems (N=107)

Topic	Programming Problems
Background problems	Background Problems (3)
Variables and expressions	Simple variables and constants (9)
	Expressions (3)
Data structures	Arrays (6)
	Collections other than arrays (2)
Input/Output	Main topic (2)
Control structures	Conditional Control Structure (11)
	Iterative Control Structure (10)
	Recursion (5)
	Exceptions (3)
	Event driven (2)
Modularization	Modularization (2)
	Function Parameters (10)
	Function returned value (3)
OO fundamentals	Classes and objects (12)
OO-design	Abstraction (11)
	Inheritance (2)
	Aggregation (1)
Problem Solving	Problem Solving (10)

As an example, Figure 2 shows the 10 programming problems related to the subtopic "Iterative Control Structure", part of the topic Control Structures.

- *Iterative control structure*
  - Inability to properly indent code with iteration statements.
  - Inability to trace the execution of loop control structures.
  - Failure to recognize the need for a loop (for example, to control potential repeated mistakes in the input of a value or using IF statements rather than loops).
  - Failure to select the most appropriate loop control structure in each context (for, while-do, do-while).
  - Improper handling of loop counter.
  - Counter variable changed in for-loop.
  - Incorrect conditions for conditional loops (for example incorrect start/termination condition, leading to off-by-one errors).
  - Incorrect update of condition in conditional loops.
  - Insertion in loop body, of code that should be executed only once before or after the loop.
  - Confusion between sequencing versus nesting of iterative statements.

Figure 2: Programming problems (N=10) related to the subtopic: Iterative Control Structure

As it can be noticed, unlike other existing lists, there is no emphasis on syntactical, language-dependent, and low-level aspects. Instead, the focus is rather on more meaningful [4, 8] conceptual and strategic knowledge. The complete list is available in the Appendix A.

5 DISCUSSION - USING THE LIST TO ENHANCE TEACHING AND LEARNING

Related to **RQ1**, the final list presents the main common topics and programming problems faced by programming novices in the fundamental areas of procedural and object-oriented programming, regardless of the programming language.

The list is grounded on the extensive literature, on the personal experience of the authors, on the critical assessment of additional experienced educators and, even if to a limited extent so far, on experimental activities with the students. Despite this, the list cannot be considered final. There is no doubt that the problems listed are real problems, yet we cannot claim that they are all perfectly language independent, that they are formulated at the ideal level of detail, or even just described in the best possible way. While it is expected, and hoped, that the list will be further refined and especially extended to other programming paradigms, Qian and Lehman [1], among others, recommend that such a list be considered a starting point, but the focus of any effort must ultimately be on the use of the list to improve pedagogical strategies.

This is exactly the goal of **RQ2**, which aims to identify how the list of common problems could be used to enhance teaching and learning. In the following subsections, indeed, we discuss the possible uses of the list elements with this goal.

5.1 Grading, developing of target tests, and evaluation rubrics

The proposed list of problems can be used, directly, to support the assessment of programming students. First, as a checklist, it can support educators to better grade students' work [6]. But the list can be conveniently used also to develop tests explicitly targeting one or more problems, as well as the associated detailed evaluation rubrics.

Figure 3 shows an example of a focused test related to the "Selection control structure" topic, with the related evaluation rubric.

Simplify the following program (that is, eliminate potential useless code):

```
BEGIN
    Read (A, B)
    IF (A>=B)
        THEN
            Write (A+B)
        ELSE
            IF (A<B)
                THEN
                    Write (B-A)
            ENDIF
    ENDIF
END
```

item-evaluation grid	Ok – y/n
Redundant use of an IF statement, instead of an ELSE clause	

Figure 3: Sample test and related evaluation rubric

Following a mastery learning approach as advocated by Luxton-Reilly et al. [17], the tests can be focused on a specific problem, such as in Figure 3. Alternatively, depending on the desired level of complexity, tests can probe several problems together: the associated evaluation rubric, precisely based on the targeted problems, supports the evaluators in diagnosing potential students' weaknesses with precision and objectivity, increasing discrimination power. Each entry could be scored dichotomously to further increase objectivity.

Obviously, the list of problems can also be conveniently used to develop detailed evaluation rubrics for pre-existing tests. And finally, the list could be used to design Concept Inventory language independent questions, where each wrong choice is a distractor mapped to a specific programming problem.

5.2 Suggesting pedagogical strategies: antipattern cards

The list has been documented (partially, so far) with antipattern cards [19]. Antipattern cards represent a considerable step beyond the simple identification of a specific problem: not only they provide further information about the problem and how to diagnose it, but they also provide examples and suggestions for pedagogical interventions, as recommended by Qian and Lehman [1].

Figure 4 shows an example of an antipattern card, corresponding to the list item "Confusion between IF <COND> and ON/WHEN <COND>" constructs. This item, as many others indeed, is by construction strongly related to issues broadly discussed in the literature. This can be considered an instance of the parallelism bug discussed by Pea [3], also mentioned in the antipattern card.

Name	Confusion between IF <COND> and ON/WHEN <COND> (parallelism bug)
Category	Control-flow / parallelism bug
Description	Student believes that when <COND> becomes TRUE, the IF statement gets executed immediately independently of the actual current instruction being executed.

Example	REPEAT FOREVER {
	Show (image1)
	Pause (time)
	Show (image2)
	Pause (time)
	IF (key A is pressed) { // key A is pressed and immediately released
	time = time * 2 // slow down animation
	}
	}

Reference to the literature	This is an instance of the "parallelism bug" identified by Pea [3] "*The parallelism bug is revealed in diverse contexts, but its essence is the assumption that different lines in a program can be active or somehow known by the computer at the same time, or in parallel.*" Possible confusion with the "ON <event>" construct typical of event-driven languages, where the condition of the ON clause is indeed constantly monitored.
Intervention suggestions	Invite to trace the execution of the following code snippets, and discuss:

Snippet A:	Snippet B:	Snippet C:

A←3
B←5
Write ("starting…")
A=A+B
B=B+A
A=A+B
B=A+B
IF (B==13) {
Write ("triggered!")
}
	A←3
	B←5
	Write ("starting…")
	IF (B==13) {
	Write ("triggered!")
	}
	A=A+B
	B=B+A
	A=A+B
	B=A+B
		A←3
		B←5
		Write ("starting…")
		A=A+B
		B=B+A
		A=A+B
		B=A+B
		- - - - -
		ON (B==13) {
		Write ("triggered!")
		}

Figure 4: Example of antipattern card for an item in the list

This problem arises when novices incorrectly believe that the condition of an IF statement is continuously evaluated, so that the body of the THEN clause is executed as soon as the condition becomes true, independently from the currently executing instruction. In this case, the origin of the problem identified by Pea [3], that is the interference with the natural language, has been enriched with an additional potential origin which was observed in our experimental activities, that is the possible confusion with the "ON <event>" construct typical of event-driven languages. In this last case, the condition of the ON construct is indeed constantly monitored, in order to trigger the execution of its body as soon as feasible.

We observed that this problem occurs more frequently when the students have been introduced first to event-driven languages. The antipattern synthetically provides a description of the problem, suggestions about how to detect it, its potential origins, and explicit suggestions on how to help students overcome it.

5.3 Additional potential Use Cases for the list

There are many other potential use cases for the list, including the following ones for which some episodic positive feedback has been obtained:

- The list was used quite creatively by one of the teachers who was asked to evaluate it. He assigned his students a task of a couple of hours about inheritance. Then he randomly and anonymously picked up a few of their submissions and publicly discussed some of their errors, pointing them to the corresponding error items in the list. He then handed over the list to the students, asking them to mark with a cross the errors on the list that they committed. Finally, he checked if they could correctly identify their own mistakes, with the objective to quickly singling out the students who had even failed to recognize their errors. The teacher expressed appreciation for the list, which allowed him to quickly identify the students with weak metacognitive skills, thus requiring special attention to support their progress.
- The list can be used as self-support material, to help students focussing their attention on potential pitfalls, and enhance their metacognition capabilities. We have observed, for example, that a few students who were provided the list could considerably reduce the number of errors that they previously unknowingly introduced while coding.
- The list could finally be used to support the set up of automatic correction systems, focusing on potential pitfalls. Besides, these systems could later be used to automatically collect data, at scale, useful for further evaluation and improvement of the list itself.

5.4 Threats to Validity

The main threat to validity of this research is that, as explained in the methodology section, an important input in the design process was the personal experience of the researchers. Therefore, if other researchers would replicate this study, even considering the same related work as source base, the list generated would be possibly partially different from the list presented (Table 2 and Appendix A) in this research, because their personal and background experiences would be different. In particular, while there are no doubts that the errors reported in the list are indeed actual possible errors, they could be formulated in a different way, especially at different levels of abstraction (that is, more or less detailed).

Nevertheless, the list is strongly based on the existing literature, subsuming other existing lists, it was evaluated through the correction of the activities of 52 students, and assessed by five additional experienced educators. As already mentioned, the list is open to further improvements and extensions, as foreseen in future activities.

Another threat to validity is that, despite the explanations in Section 4, the list presented in this research does not show an exact tracking, for each item, of how it was generated and exactly which literature elements (if any) that item was based on/derived from. Some of this information, however, is available in the Antipattern Cards, as exemplified in Figure 4.

Finally, the paper suggests several strategies to make use of the list to improve pedagogical activities. While the experience of the authors and first anecdotal evidence support the validity of these proposals, additional larger scale empirical studies with educators and students are certainly required, and already being carried out, although still at an early stage.

6 ROAD TO THE FUTURE

6.1 Self-sustainability of the open list of common problems

The list in the Appendix draws on a lot of literature and experience, yet it cannot be considered casted in stone. To encourage its possible improvement, extension, and specialization, as well as its practical use in teaching, it is going to be published as an Open Educational Resource, with a CC BY-SA-NC 3.0 license in the OER Commons and Merlot repositories. We ourselves are planning other activities that make use of the proposed list while possibly further improving it, and we will be monitoring possible derivatives and be happy to integrate feedback from other practitioners and researchers. Hopefully, this will make it possible to improve the list and keep it alive with a self-sustainable process.

6.2 Item banking: battery of crowdsourced, standardized, calibrated tests

So far, we have made abundant use of open-answer tests in our activities, but we would like to make more extensive use of precisely targeted Multiple-Choice Tests (MCT) because they are more objective, easier to score, and easily automated, hence more easily integrated in teaching practice, and at scale.

Therefore, as a future activity, we plan to develop and openly publish an initial seed set of MCT tests, along the lines of the Canterbury Question Bank [6], but experimentally validated and calibrated. The validity of the tests would be assured by grounding them on the list of problems, through expert consensus and experimentation. The use of targeted tests and detailed evaluation rubrics as previously discussed, would make students' evaluation more straightforward and objective, increasing – in particular – inter-rater reliability [17]. The tests' reliability would be further supported by the experimental analysis of test results with the Item Response Theory (IRT) [20] – initially the Rash model because it is more suitable for smaller samples.

The use of IRT makes it possible to estimate the difficulty of each test on a common scale: this would allow us to incrementally integrate additional tests, aligning ("equating") them on the same difficulty scale. The possibility to incrementally integrate further tests to the initially provided seed tests, would open up the possibility to crowdsource additional tests, aiming to obtain an extensible crowdsourced open battery of standardized, calibrated tests ("item banking"). This would be facilitated by publishing the initial seed tests together with the methodology to develop and calibrate additional ones.

6.3 Personalized evaluation and reliable comparisons

A plus of the strategy just outlined, is that these calibrated tests would be also suitable to be delivered, at scale, with computerized adaptive evaluation systems [21]. These systems would automatically select, from the test battery, those tests that more precisely match the level of ability of the tested students, thus increasing the accuracy of their personalized evaluation on a common scale.

Additionally, the availability of a battery of standardized/calibrated tests would make it possible to reliably compare different pedagogical strategies or educational systems, one of the goals of CIs. Yet in this case, unlike in the case of CIs, a large test battery would not suffer from the problem of saturation.

7 CONCLUSIONS

This paper has described the development of a comprehensive – although not final – open list of problems commonly experienced by novice programmers, as language agnostic and conceptually and strategically oriented as possible. It has exemplified the use of the proposed list to improve the evaluation of students in the teaching practice, with targeted tests and detailed evaluation rubrics. Items in the proposed list have been documented with antipattern cards, which not only provide examples on how to detect the specific problem, but also provide indications on its possible origin, and suggestions for pedagogical strategies to overcome it.

The paper finally outlines a strategy to crowdsource a battery of standardized/calibrated tests, that can be used both for the formative and summative evaluation of students, as well as to objectively compare different educational strategies and educational systems.

8 ACKNOWLEDGEMENTS

We would like to express our gratitude to the educators who gave us their expert advice on the proposed list, to Dr. Paul Mulholland for the stimulating discussions with one of us, and to the anonymous reviewers for their useful input.

REFERENCES

[1] Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. ACM Transactions on Computing Education (TOCE), 18(1), 1-24.

[2] Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010, March). Identifying student misconceptions of programming. In Proceedings of the 41st ACM technical symposium on Computer science education (pp. 107-111).

[3] Pea, R. D. (1986). Language-independent conceptual bugs in novice programming. Journal of Educational Computing Research, 21, 25-36.

[4] Mccauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C., (2008). Debugging: A review of the literature from an educational perspective. Computer Science Education. 18. 10.1080/08993400802114581.

[5] Holland, S., Griffiths, R., & Woodman, M. (1997, March). Avoiding object misconceptions. In Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education (pp. 131-134).

[6] Sanders, K., & Thomas, L. (2007). Checklists for grading object-oriented CS1 programs: concepts and misconceptions. ACM SIGCSE Bulletin, 39(3), 166-170.

[7] Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the scope of concept inventories for introductory computing subjects. ACM Transactions on Computing Education (TOCE), 10(2), 1-29.

[8] Hanks, B. (2007). Problems encountered by novice pair programmers. In S. Fincher, M. Guzdial & R. Anderson (Eds.), Proceedings of the 3rd international computing education research workshop (pp. 159–164). New York: ACM Press.

[9] Tew, A. E., & Guzdial, M. (2011, March). The FCS1: a language independent assessment of CS1 knowledge. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 111-116).

[10] Parker, M. C., Guzdial, M., & Engleman, S. (2016, August). Replication, validation, and use of a language independent CS1 knowledge assessment. In Proceedings of the 2016 ACM conference on international computing education research (pp. 93-101).

[11] Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2017). A basic recursion concept inventory. Computer Science Education, 27(2), 121-148.

[12] Porter, L., Zingaro, D., Liao, S. N., Taylor, C., Webb, K. C., Lee, C., & Clancy, M. (2019, July). BDSI: A validated concept inventory for basic data structures. In Proceedings of the 2019 ACM Conference on International Computing Education Research (pp. 111-119).
[13] Farghally, M. F., Koh, K. H., Ernst, J. V., & Shaffer, C. A. (2017, March). Towards a concept inventory for algorithm analysis topics. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 207-212).
[14] Caceffo, R., Wolfman, S., & Booth, K. (2016). Developing a Computer Science Concept Inventory for Introductory Programming. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). ACM, New York, NY, USA, 364-369. DOI=http://dx.doi.org/10.1145/2839509.2844559
[15] Pillay, N., & Jugoo, V. R. (2006). An analysis of the errors made by novice programmers in a first course in procedural programming in Java. In Proceedings of the 36th SACLA Conference (pp. 84-93).
[16] Robins, A., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. In Proceedings of the 8th Australasian computing education conference (pp. 165-173). Hobart: Australasian Computer Society.
[17] Luxton-Reilly, A., Becker, B. A., Cao, Y., McDermott, R., Mirolo, C., Mühling, A., ... & Whalley, J. (2018, January). Developing assessments to determine mastery of programming fundamentals. In Proceedings of the 2017 ITiCSE Conference on Working Group Reports (pp. 47-69).
[18] Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. Inroads, 35(1), 153-156.
[19] Koenig, A. (March-April 1995). "Patterns and Antipatterns". Journal of Object-Oriented Programming, 8 (1), 46-48.
[20] Yu, C. H. (2013) A simple guide to the item response theory (IRT) and Rasch modeling. Retrieved from www. creative-wisdom. com/computer/sas/ IRT.pdf.
[21] Choi, Y., & McClenen, C. (2020). Development of adaptive formative assessment system using computerized adaptive testing and dynamic bayesian networks. Applied Sciences, 10(22), 8196.
[22] Gama, G., Caceffo, R., Souza, R., Benatti, R., Aparecida, T., Garcia, I., & Azevedo, R. (2018). An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python. In Technical Report 18-19, Institute of Computing, University of Campinas, SP, Brasil. 106 pages. November, 2018.
[23] Caceffo, R., Frank-Bolton, P., Souza, R., & Azevedo, R. (2019). Identifying and Validating Java Misconceptions Toward a CS1 Concept Inventory. In Innovation and Technology in Computer Science Education (ITiCSE '19), July 15-17, 2019, Aberdeen, Scotland UK. ACM, New York, NY, USA.
[24] Brown, N., & Altadmri, A. (2017). Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. ACM Transactions on Computing Education.
[25] Izu, C., Ng, D., & Weerasinghe, A. (2022). Mastery Learning and Productive Failure: Examining Constructivist Approaches to teach CS1. In Proceedings of the 33rd Annual Workshop of the Psychology of Programming Interest Group (PPIG), Milton Keynes, United Kingdom.

APPENDIX A – LIST OF COMMON PROBLEMS

A.1 – Background Problems

- *Background Problems*
  - Basic syntax errors.
  - Wrong basic structural details (e.g., data outside classes, code outside methods…).
  - Difficulties in using or setting-up the development environment.

A.2 – Variables and Expressions

- *Simple variables (and constants)*
  - Meaningless or misleading variable names, reflecting lack of clarity about their purpose (including using the same variable with different roles).
  - Difficulty to differentiate among name, value, and address of a variable (for example, variables with names that are actually their possible values.)
  - Missing or unnecessary variable declaration (when applicable).
  - Incorrect definition of variable type (when applicable).
  - Missing/incorrect/unnecessary variable initialization.
  - Overwriting the content of a variable before using it.
  - Attempt to change the value of a constant.
  - Failure to understand scope-rules.
    - Attempt to access local variables from outside their scope (for example attempting to access local variables belonging to functions in the call stack or declaring variables inside a block and trying to access them from outside the block.
    - Global variables considered as local in the current scope.
    - Failure to grasp that global variables are accessible from within a method.
    - Unhealthy use of global variables.
  - Failure to understand variables' lifetime (thinking, for example, that a standard local variable in a subprogram keeps its value between different calls).
- *Expressions*
  - Type mismatch in expressions.
  - Wrong order / precedence of operators in expressions (including, for example, misuse of parenthesis).
  - Misuse of logical operators in expressions.

A.3 – Data structures

- *Arrays*
  - Failure to recognize the opportunity to use arrays.
  - Confusing cell index and cell content.
  - Confusing the single cell and the whole array.
  - Considering the array as a simple (single/primitive value) variable (e.g. in assignments, copy, or comparisons).
  - Failure to identify the opportunity to use parallel and/or multi-dimensional arrays.
  - Incorrect use of indexes (including in parallel and multi-dimensional arrays).
  - Incorrect array declaration (frequently its dimension).
- *Collections other than arrays*
  - Inability to select and justify the most appropriate data structure in a given context (for example a stack versus a list).
  - Inability to justify the most suitable implementation of a data structure for a given context (e.g. static versus dynamic, single versus double linked).

A.4 – Input/Output

- *Main Topic*
  - Inability to identify the input or the output of a program.
  - Inability to use the input/output mechanisms available in the target language (for example, confusing file open/read/write versus file redirection).

A.5 – Control Structures

- *Conditional control structures*
  - Inability to properly indent code with IF statements.
  - Inability to manually trace the execution of IF statements.
  - Failure to recognize the opportunity to use selection statements (e.g. IF, or Switch).
  - Confusion between IF <COND> and ON/WHEN <COND> (parallelism bug)
  - Redundant use of an IF statement, instead of an ELSE clause.
  - Code repeated both in the THEN and ELSE clauses.
  - Redundant structured if-statements where boolean expression could simplify the code.
  - Confusion between sequencing versus nesting of IF-statements.

- Multiple IF statements fail to cover all the necessary cases.
- Redundant use of conditions in structured IF-statements.
- Unreachable statement.

- *Iterative control structure*
  - Inability to properly indent code with iteration statements.
  - Inability to trace the execution of loop control structures.
  - Failure to recognize the need for a loop (for example, to control potential repeated mistakes in the input of a value or using IF statements rather than loops).
  - Failure to select the most appropriate loop control structure in each context (for, while-do, do-while).
  - Improper handling of loop counter.
  - Counter variable changed in for-loop.
  - Incorrect conditions for conditional loops (for example incorrect start/termination condition, leading to off-by-one errors).
  - Incorrect update of condition in conditional loops.
  - Insertion in loop body, of code that should be executed only once before or after the loop.
  - Confusion between sequencing versus nesting of iterative statements.
- *Recursion*
  - Inability to manually trace the execution of a recursive method.
  - Failure to conceive a recursive solution, insisting on an iterative one.
  - Lack of recursive method invocation.
  - Incorrect computation of the return value of a recursive method.
  - No termination at base case (because base case not specified, or because never reached).
- *Exceptions*
  - Inability to trace the execution of code with Exceptions.
  - Inability to use (throw or catch) existing Exceptions (confusing, for example, code that should throw an exception, and code that should catch it).
  - Inability to develop custom Exceptions.
- *Event driven*
  - Inability to trace the execution of code with event-driven software.
  - Inability to conceive reactive event-driven programs, insisting on pro-active software programming style.

A.6 – Modularization

- *Modularization*
  - Inability to trace the execution of code with subprograms.
  - Inability to restructure, simplifying it, complex monolithic code.
    - Inability to identify meaningful blocks of codes suitable to be abstracted as subprograms (for example by writing the same code multiple times).
    - Inability to structure the subprograms in layers of *homogeneous* levels of abstraction.
- *Function parameters*
  - Missing/incorrect declaration of formal parameters (when applicable).
  - Logic error in providing actual parameters in function invocation (including, for example, missing actual parameters).
  - Incorrect order of actual parameters in function invocation.
  - Incompatible types between formal and actual parameters.
  - Overwriting the value of a parameter before using it.
  - Assigning a parameter to a redundant variable inside the function.
  - Actual parameters not used in the function's body.
  - Confusion between parameter and same-name variables.
  - Parameters considered accessible outside their scope.
  - Confusion between passing by value versus passing by reference.
- *Function returned value*
  - No value returned by a function that should return one (for example, the function visualizes a value rather than returning it).
  - Value returned by a function incorrectly ignored in the invoking context.
  - Type mismatch between returned value and its use in the caller.

A.7 – OO fundamentals

- *Classes and Objects*
  - Confusion among declaration, instantiation, and use of an object.
  - Use of object attributes or methods before instantiating the object.

    - Re-declaration of an object (Object name preceded by its class name after previous declaration).
    - Handling objects as simple variables.
    - Unnecessary instantiation of objects.
    - Using method parameters when object attributes should be used, and vice versa.
    - Confusing class, and instance variables.
    - Confusing instance, and local variables (for example, re-declaring attributes as local variables in constructors, leading to failed initialization).
    - Confusing class, and instance methods.
    - Instance / class conflation (classes identical except minor variations, classes never instantiated more than once, superclass/subclass used instead of class/instance)
    - Improper use of current (this / self / Me) object.
    - Conflation between an object and its member variables.

A.8 – OO Design

- *Abstraction*
    - Inability to identify classes modeling objects in the domain.
    - Inability to identify the attributes representing the state of an object.
    - Inability to identify the methods representing the behavior of an object (that is, determining which methods should be in the public interface).
    - Inability to define the appropriate signature of methods.
    - Classes defined but not used.
    - Confusion concerning the identification of suitable parameters for instance versus class methods.
    - Inconsistent naming of equivalent methods/variables in different classes (problems with encapsulation).
    - Inability to identify suitable constructors.
    - Inability to make use of polymorphic methods.
    - Duplicated method signatures in different classes not defined as interfaces (Java specific).
    - Inability to organize the overall structure of a program, in terms of interacting objects.
- *Inheritance*
    - Failure to use inheritance to model hierarchical domains.
    - Confusion between inheritance and aggregation.
    - Inability to use the inheritance technical mechanisms of the target language.
- *Aggregation*
    - Code in single class instead of composite class and parts.

A.9 – Problem Solving

- *Problem Solving*
    - Not knowing how to get started or organize a solution to a problem.
    - Difficulties in understanding the problem (including, for example, inability or reluctance to simulate the problem by hand).
    - Difficulties in reformulating the problem.
    - Inability to identify input and output (repeated entry).
    - Inability to identify proper test patterns.
    - Inability to verify whether the solution provided complies with the assigned task.
    - Stuck on program design (solution understood but can't turn that understanding into a program).
    - Inability to break the proposed problem into smaller subproblems.
    - Inability to redefine the proposed problem to make it more like other problems whose solution is already known.
    - Inability to simplify the proposed problem to start synthesizing the nucleus of a first solution.
    - Inability to generate multiple tentative solutions.
    - Inability to critically analyze the alternative tentative solutions.