Automated Assessment of Learning Objectives in Programming Assignments

Arthur Rump[0000−0002−4880−4994], Ansgar Fehnker[0000−0002−5326−3432], and Angelika Mader[0000−0002−7065−2640]

University of Twente, Enschede, The Netherlands
hello@arthurrump.com, ansgar.fehnker@utwente.nl, a.h.mader@utwente.nl

Abstract. Individual feedback is a core ingredient of a personalised learning path. However, it also is time-intensive and, as a teaching form, it is not easily scalable. In order to make individual feedback realisable for larger groups of students, we develop tool support for teaching assistants to use in the process of giving feedback. In this paper, we introduce Apollo, a tool that automatically analyses code uploaded by students with respect to their progression towards the learning objectives of the course. First, typical learning objectives in Computer Science courses are analysed on their suitability for automated assessment. A set of learning objectives is analysed further to get an understanding of what achievement of these objectives looks like in code. Finally, this is implemented in Apollo, a tool that assesses the achievement of learning objectives in Processing projects. Early results suggest an agreement in assessment between Apollo and teaching assistants.

Keywords: Programming Education · Automated Assessment · Automated Feedback

1 Introduction

Learning in the perspective of the 21st-century skills aims, among others, at enthusiasm, deep understanding, the ability to apply, and reflection. For the programming courses of our program of Creative Technology, we address these skills by giving open assignments that allow for individual solutions and creativity, while making students owner of their learning process. The programming assignments let students define their own project, as long as they use the concepts taught in the course and demonstrate mastery of the learning outcomes.

A driving principle in this personalised learning process is individual feedback to get students unstuck in their learning path when needed. However, individual feedback is time-intensive, and, accordingly, does not scale well with an increasing number of students. In order to make individual learning processes also realisable for larger groups of students, we develop tools that support teaching assistants in giving feedback.

We developed an online platform called Atelier to aid communication between students and teaching assistants during programming tutorials. Students

can upload their code and share it with teaching assistants. Teaching assistants can leave comments and see each others' comments, which contributes to consistency in giving feedback. Additionally, an automated code checker [6] identifies standard faults, suggesting comments that the teaching assistant can share and discuss with students. This increases the efficiency of giving feedback.

This paper introduces the extension *Apollo*[1], which analyses programs submitted to Atelier with respect to the desired learning outcomes for the course. The results of Apollo help teaching assistants to identify shortcomings of a program faster, and hence also contributes to an increased efficiency and consistency in giving feedback. Atelier and Apollo were developed in the context of the engineering and design bachelor programme Creative Technology at the University for Twente, where students start programming in Processing[2].

Section 2 covers related work on learning outcomes and automated feedback. In Section 3 we investigate learning outcomes in programming courses and how mastery of these learning outcomes can be identified by an automated tool is described in Section 4. Section 5 uses historical data to calibrate the system and provides early validation of the approach when used in an actual course. The final section closes with discussion and conclusions.

2 Background

This section introduces learning outcomes, including characteristics related to their suitability for automated assessment, and approaches to automated feedback. The combination of both forms the basis for Apollo.

Learning outcomes. Learning outcomes range from vague aspirations to achieve at the end of a program to very specific objectives to accomplish in a single lecture. Wilson [16] splits learning outcomes into aims, goals and objectives.

- *Aims* give a general direction and are not directly measurable. They are meant to guide an entire program or subject area. An example from our programme is "Graduates understand and can use technology in the domain of software, algorithms and physical interaction."
- *Goals* are more specific than aims in terms of scope, but they can still relate to an entire program or subject area. They can be formulated as a concrete action, but do not have to be. An example from our program is "Students can create algorithms for solving simple problems."
- *Objectives* are often written in behavioural terms to describe more specific learning outcomes. They should be observable and measurable, like "Students can implement a divide-and-conquer algorithm for solving a problem."

We use the term 'learning outcome' to mean the intended learning outcome of a course, which may or may not be the actual learning outcome for a student.

[1] Available via https://github.com/creativeprogrammingatelier/apollo
[2] See https://processing.org

Learning outcomes can also be categorised according to "levels of mastery", similar to Bloom's Taxonomy [3,10]. The Computer Science curriculum guidelines by ACM and IEEE [1] use three levels:

- *Familiarity* means the student knows a concept, or the meaning of a concept, but is not able to apply it.
- *Usage* means whether the student can concretely use a concept, for example in a program or when doing analysis.
- *Assessment* means that the student can argue for the selection of a concept to use when solving a problem. This also requires the student to understand available alternatives.

When looking at programming assignments, the related learning outcomes are usually at the 'usage' or 'assessment' level.

Automated feedback A common approach that has been used since the 1960s is the use of automated testing tools to check student submissions for correctness. Douce, Livingstone and Orwell [5] describe several generations of these tools, which all have in common that they require well-defined exercises with supplied test cases to function correctly. In our context where students choose their own projects to work on, these tools are not applicable.

Keuning, Jeuring and Heeren [9] reviewed 101 tools, and found a total of 8 approaches to provide feedback to students, of which automated testing is just one. Four of the other approaches are specific to programming: external tools (such as compilers), static analysis, program transformations, and intention-based diagnosis.

This last approach tries to uncover the student's strategy by matching code with known ways and code patterns to achieve (sub)goals, following a structured approach to programming [15]. A tool that uses this strategy is PROUST [7,8], which was developed to provide feedback on free-form programming assignments, based on goals the students learned approaches for. This suggests that a similar approach would work well in our context of creative assignments.

An example of a tool that tracks the progress of students is the ACT Programming Tutor (APT) [4], which has a Skill Meter that shows the probability that the student has mastered that skill. APT is also goal-oriented, but works with production rules based on the current assignment: it will not let students take steps that are not on a known path to a correct solution. While the Skill Meter serves a goal similar to what Apollo aims to achieve, the APT approach is not applicable in our context: all assignments need predefined solutions, whereas our course has requirements that allow for a variety of individual solutions.

3 Learning Outcomes

The goal of this section is to identify the learning outcomes that are *assessable* by an automated tool. First, those common in computer science courses are investigated, then those specific to our course.

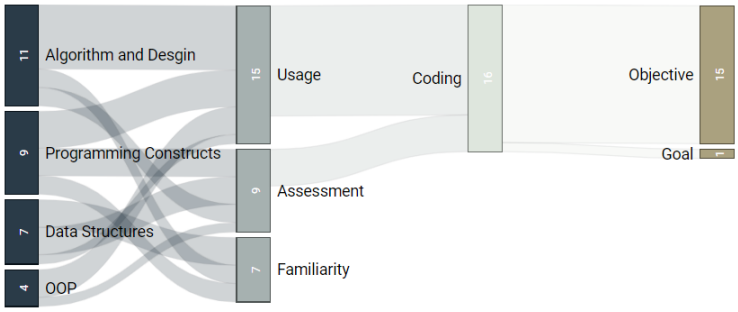

Fig. 1. Leaning outcomes from the knowledge areas Software Development Fundamentals (Algorithms and Design, Programming Concepts, and Data Structures) and Programming Languages (OOP) in the ACM and IEEE curriculum guidelines.

Fig. 1 illustrates the selection criteria applied to 31 learning outcomes related to programming in the ACM and IEEE Computer Science curriculum guidelines [1], from four topic areas relevant to our course. First, they are distinguished by their level. Only the 'usage' and 'assessment' level are relevant for our purpose since 'familiarity' precludes the ability to use a concept by definition. Next, the learning outcomes need to be directly related to writing code. And finally, the learning outcomes should be concrete and observable, which means that only *objectives* are assessable. Considering these criteria, 15 out of 31 analysed learning outcomes were found suitable for automated assessment.

The focus of our course lies on using programming as a tool for expressing creative ideas [11]. This involves, for example, simulating basic but specific physical systems, rather than learning algorithms in isolation, as would be common in classic computer science programming courses. Consequently, the official learning outcomes have a clear focus on the 'usage' level. The specific learning objectives, however, are similar to other introductory programming courses: students have to learn how to write for-loops, they have to understand how variables work and apply basic object-oriented programming concepts.

Based on the curriculum guidelines [1], the actual learning outcomes of our course and the textbook [14], we defined five representative learning objectives:

1. *Write a program that uses graphical commands to draw to the screen.*
2. *Write a program that uses looping constructs for repetition, using the appropriate looping construct.*
3. *Compose a program using classes, objects and methods to structure the code in an object-oriented way.*
4. *Implement message passing to enable communication between classes in a complex program, instead of using global variables.*
5. *Use elementary vector operations to simulate physical forces on an object.*

These objectives are selected to give a fair representation of different types of learning objectives and different degrees of freedom in the resulting code. The

five topics are a selection of what students are expected at the end of the course, which also concludes the first year of the Creative Technology programme.

4 Recognising Learning Objectives

Given the learning outcomes suitable for automated analysis, as identified in the previous section, this section focuses on the specific analysis techniques. We first describe the general technology used to detect evidence, and then for each of the five learning objectives what will count as evidence of mastery.

Finding evidence means to recognise relevant usage of programming constructs and programming patterns. Previous research on static analysis of Processing projects [2,6] successfully adapted PMD[3]. It is also used for Apollo.

For most rules, Apollo uses the following function to translate the count of occurrences to a probability of showing convincing evidence:

$$S_{a,b}(n) = 1 - \frac{1}{\frac{1}{a}n^b + 1} \tag{1}$$

This function defines a family of "S"-shaped curves in the range $[0, 1\rangle$, where the slope is determined by the parameters a and b, which can be varied from objective to objective, and from course to course.

The remainder of this section describes which aspects are relevant for the different learning objectives, and how occurrences of relevant structures in the code are counted. Unless mentioned otherwise, the S function is used to translate this count into a probability. Its parameters will be determined in Section 5.

1. *Write a program that uses graphical commands to draw to the screen.*
   This seems like a relatively simple goal, but full mastery also means that students know different methods, can familiarise themselves with methods that were not explicitly covered in the course and are able to use advanced concepts, such as affine transformations.
   Apollo uses a list of all graphical commands in Processing [12], grouped by category. For a given program it creates a list of graphical commands that have been used. The different metrics are then calculated as follows:
   (a) *Use of a variety of different drawing methods covered in the course.*
   Count the number of method calls from categories covered in the course.
   (b) *Use of advanced drawing methods, like those in the transform category.*
   Count the method calls in the transform category of drawing methods.
   (c) *Use of methods that are not explicitly part of the course material.*
   These are the methods that were not part of the count for the first metric.
   The probability for the entire learning objective is a weighted average of the individual probabilities. The first aspect has weight 3, the second weight 2 and the last weight 1. Students who do not use the covered drawing functions frequently are unlikely to master the drawing commands, even if they do use advanced or non-covered methods, so the first aspect should weigh most.

[3] PMD is a tool for detecting code smells in Java, available at https://pmd.github.io

```
for(int i = 0; i < ts.length; i++){
  Thing t = ts[i]; // Get an element
  s += i * t.getValue(); // Use index directly
}
```

Fig. 2. Example of an array iteration that requires the use of an index.

2. *Write a program that uses looping constructs for repetition, using the appropriate looping construct.*
   This objective has two aspects: usage of loops, and choosing the appropriate type of loop for a goal. In the context of our course we distinguish the following goals with the related code patterns:
   - Repeating some code while a condition holds, using a while-loop.
   - Repeating a task n times while increasing a counter, using a while- or for-loop. For-loops are preferred in this case.
   - Iterating over all items in an array, using a for-, while-, or foreach-loop; the latter is the preferred option.
   - Iterate over all items in an array while using the index independently. In this case, a foreach-loop can not be used, and the for-loop is preferred. See Fig. 2 for an example.

   Instead of listing and matching on every possible coding pattern related to using loops, Apollo characterises loops based on their usage. This includes the type of looping condition used, the types of variables used in the body and how the iterator variable is used. Based on this, three metrics are calculated as follows:
   (a) *Use of different types of loops.* The number of different types (either for, while, or foreach) of loops used in the program.
   (b) *Use of loops in a variety of situations, e.g. to iterate over an array, but also for simple repetitions.* Count the number of occurrences of loops with different characterisations.
   (c) *Choose the appropriate looping construct for a given task.* This is calculated as the ratio of loops that are the most appropriate in that situation over all used loops. This value already expresses the chance that a correct looping construct is chosen, and does not need to be converted.

   The probabilities resulting from these metrics are averaged to calculate the probability that the program contains convincing evidence that the student achieved this learning objective.
3. *Compose a program using classes, objects and methods to structure the code in an object-oriented way.*
   This goal is not just about using the keyword class, but about structuring the program by grouping related data and actions together. The paper [6] proposes an object-oriented structure for interactive Processing applications and also provides static analysis rules for automated detection of so-called design smells in this structure. If a program is structured into multiple classes

and none of the common design smells is detected, the student has likely mastered this objective. Also, it is assumed that any useful class has methods; Apollo uses a minimum of 2.

The metrics are calculated as follows:

(a) *Use of classes with various methods.* This is simply the number of classes with more than two methods.

(b) *Relatively few detected design smells in the code patterns.* For this metric, Apollo runs the static analysis rules defined in [6] to detect code smells in the program. Because the chance of a small mistake is bigger in a large program, the amount of smells is divided by the number of classes counted for the first metric. In this case, since a larger number means more problems, and less evidence of mastery, it uses a flipped version of function S.

The probability for the overall objective is the average of these two metrics.

4. *Implement message passing to enable communication between classes in a complex program.*

   This objective is related to the previous objective of object-oriented design but is included separately since it receives dedicated attention in the course materials. The goal is to share information between classes. A common, but discouraged, practice is to define a global variable that several objects use. The alternative is method parameter passing, where one object calls a method of another object and passes the information by parameter. The second option is preferred because the information is not shared with other parts of the program that might inadvertently change its value.

   To detect if the student can apply message passing, Apollo first finds all global variables. Only variables that are mutated in the program should count for message passing, so global constants are filtered out. Then the number of uses of these global variables across different classes is counted.

   The detection of parameter passing happens similarly: Apollo determines all calls to methods from outside the defining class and counts the passed arguments. To get a similar count to the global variable use, only arguments that are locally declared values are counted.

   The probability that the program contains convincing evidence that the learning objective is achieved, is then calculated as the ratio of parameter passing instances over the total count of both communication methods.

5. *Use elementary vector operations to simulate physical forces on an object.* Forces, acceleration, velocity and position are modelled in Processing using the `PVector` class, so physical formulas are commonly translated into operations on these vectors. It is possible to define code patterns for common goals, such as “apply a force to an object with mass”, “calculate the drag force for an object in a medium”, “model gravity using constant downward force” or “calculate friction”. If one or more of these patterns related to known goals are found, it is a good indicator for mastery of this learning objective.

   There are, however, many more physical phenomena than specifically covered in the course and known to Apollo, especially since students are free to define their own project. As a weak, but more general, indicator Apollo counts all

operations on `PVector` objects. While this could also indicate abstract linear algebra, an absence of `PVector` usage does indicate that the student is not using `PVector`s for physical modelling.
To keep things manageable, Apollo only considers operations on `PVector`s when comparing code patterns related to physics and simply ignores the calculation of other parts of the solution. Apollo recognises five code patterns for simulating physics, which are defined as a list of method calls on different `PVector` instances.
Two metrics are computed as follows:

(a) *Use of a known pattern for working with forces.* Count the number of times one of the five specified code patterns for physics matches with the code, as explained before.
(b) *Use of operations on* `PVector`*s.* Simply count the number of operations based on the declared `PVector`s.

The overall probability is calculated as the weighted average of both metrics, where the first metric has weight 1, the second 2.5. This means that even when no code patterns are recognised, a chance of 0.7 can still be reached based on the number of `PVector` operations, which is desired because of the mentioned limitations on detection of known physical structures.

5 Calibration and Validation

For an initial validation and determining the parameters for converting counted metrics into probabilities, Apollo was used on an old dataset of final student projects used in the evaluation of [6].

Calibration. To illustrate the method used for calibration of the parameters, we discuss the example of counting calls to drawing methods covered in the course materials. Other metrics were calibrated in a similar fashion.

Considering the drawing methods, we find that all programs in the dataset use between 4 and 17 different drawing methods, with a median of 9. The first quartile is at 8 drawing methods, the third quartile at 11. To determine the correct parameters for the function S, these statistics were mapped to the desired chance. The first quartile is mapped to 50%, the median to 70% and the third quartile to 95%. This means that a chance of understanding above 95% is assigned to the 25% best programs, above 70% to the 50% best programs etc. For the covered drawing methods this means that 8 used drawing methods maps to a 50% chance of being convincing evidence, 9 used maps to 70% and 11 used to 95%. The parameters were then determined with a standard curve fitting algorithm.

Integration. To test Apollo in practice, the tool was integrated with Atelier, our online platform for programming tutorials, where it creates comments with its assessment on every uploaded project. To reduce the chance that teaching assistants would interpret Apollo's assessments as grading, the probabilities calculated by Apollo were translated into words, following a mapping defined in

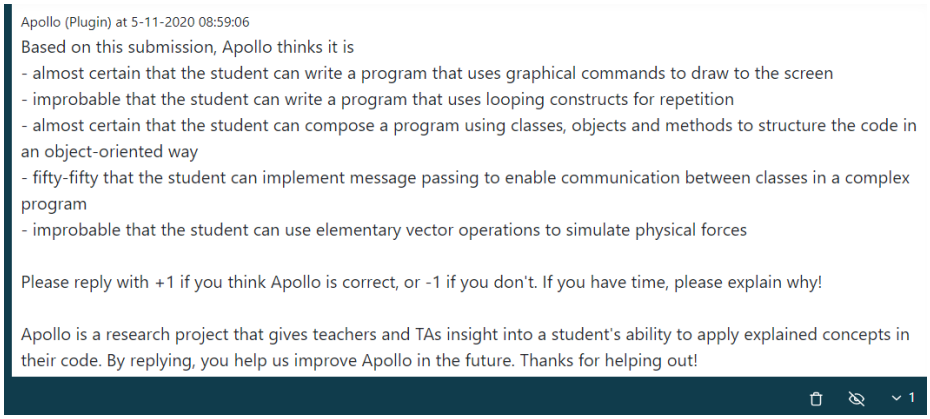

Fig. 3. An example comment by Apollo. The second icon in the lower left corner indicates that this comment is not shared with the student, but only visible for the teaching assistant.

[13]. A chance of 20% or less is rendered as 'improbable', for example. The intention is that teaching assistants use this information to address when they provide feedback, and not as a substitute for grading. See Fig. 3 for an example.

Apollo ran on 65 student submissions in the final week of the tutorial period where students could ask for feedback on their projects. On 11 of these submissions, a teacher or teaching assistant indicated whether or not they agreed with Apollo's assessment. Out of these 11 comments, 8 were positive and 3 neither positive nor negative. Message passing and physics were both mentioned three times in these comments. In the case of message passing, always because Apollo indicated a low score on these programs and the commenter agreed with that assessment. For physics this was the case for two comments; the third indicated that there was some physics in the program which was not detected by Apollo.

The low response rate can at least partially be attributed to the fact that Apollo was only deployed in the last week of tutorials when many students seek help on their final projects. Teaching assistants rightly prioritise helping as many students as possible in the limited time they have during a tutorial.

In separate interviews with two teaching assistants, both indicated to agree with the comments made by Apollo most of the time. One of them indicated that they would like to get more information on why Apollo made the assessment, the other mentioned that the overall information presentation in Atelier could be better and that it sometimes feels "rather spammy." Both did find the provided information useful when looking at a project uploaded by a student.

6 Conclusions

This paper reports on automatic assessment of learning objectives in a first-year programming course, and its implementation in Apollo. Apollo is an extension of Atelier, our online platform that supports teaching staff in the process of giving feedback to students. To develop the tool, we first selected learning outcomes suitable for automatic analysis of students' code in a systematic way, starting with the Computer Science Curriculum Guidelines, followed by the learning outcomes for programming courses in our program. In general, learning outcomes on the correct usage of e.g. programming elements are suitable for automatic analysis, while more general goals concerning the understanding of an area are not. Based on this, we identified a set of five representative learning objectives to use for the development of Apollo.

For each of these learning objectives, we identified aspects that indicate mastery of that objective and implemented rules in Apollo to recognise these aspects in code. In the end, Apollo draws from both the static analysis and intention-based diagnosis techniques for providing automated feedback. The core of intention-based diagnosis is to uncover the goal a student had in mind while writing their code by identifying common patterns used to achieve these goals. This is most prominent in the learning objective on modelling physics, where concrete patterns were used to represent common goals in that area. For other objectives, we instead tried to characterise the common solutions to achieve a goal to avoid having to list all possible ways in which a loop can be used.

While validation of the tool was limited due to time constraints, feedback from teaching assistants does indicate that Apollo is a useful addition to the Atelier platform. Early signs also tend to indicate that Apollo's assessment is mostly correct, but further validation is required to make a clear statement on this. This will happen as part of an ongoing longer-term study.

There are two main areas in which Apollo could be improved. First, Apollo only assesses individual programs, but it cannot yet accumulate outcomes from a series of programs of one student to an overall chance that a student has achieved a learning objective. Neither can it combine the results of a group of students to create an overview of the overall progress in a course. These additions would extend the use of Apollo beyond giving feedback on single programs into giving insight into the learning paths of students.

Second, instead of a textual presentation, more intuitive visualisations of Apollo's results would be desirable. This becomes especially important when the results are combined into student- and group-level numbers, possibly tracked over time. An insightful presentation of results would also be useful for an evaluation to what extent the tool does improve the efficiency of giving feedback. These two areas will be the subject of further research.

References

1. ACM Computing Curricula Task Force (ed.): Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. ACM, Inc (1 2013). https://doi.org/10.1145/2534860
2. Blok, T., Fehnker, A.: Automated program analysis for novice programmers. In: Proceedings of the 3rd International Conference on Higher Education Advances. Universitat Politècnica València (6 2017). https://doi.org/10.4995/head17.2017.5533
3. Bloom, B., Englehart, M., Furst, E., Hill, W., Krathwohl, D.: Taxonomy of educational objectives. The classification of educational goals. David McKay Co., Inc, New York (1956)
4. Corbett, A.T., Anderson, J.R.: Knowledge tracing: Modeling the acquisition of procedural knowledge. User Modelling and User-Adapted Interaction **4**(4), 253–278 (1995). https://doi.org/10.1007/bf01099821
5. Douce, C., Livingstone, D., Orwell, J.: Automatic test-based assessment of programming. Journal on Educational Resources in Computing **5**(3), 4–es (9 2005). https://doi.org/10.1145/1163405.1163409
6. Fehnker, A., de Man, R.: Detecting and Addressing Design Smells in Novice Processing Programs, pp. 507–531. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-21151-6_24
7. Johnson, W.L., Soloway, E.: Intention-based diagnosis of programming errors. In: Proceedings of the 5th National Conference on Artificial Intelligence, Austin, TX. pp. 162–168 (1984)
8. Johnson, W., Soloway, E.: Proust: Knowledge-based program understanding. IEEE Transactions on Software Engineering **SE-11**(3), 267–275 (3 1985). https://doi.org/10.1109/tse.1985.232210
9. Keuning, H., Jeuring, J., Heeren, B.: A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education **19**(1), 1–43 (1 2019). https://doi.org/10.1145/3231711
10. Krathwohl, D.R.: A revision of Bloom's taxonomy: An overview. Theory Into Practice **41**(4), 212–218 (11 2002). https://doi.org/10.1207/s15430421tip4104_2
11. Mader, A., Fehnker, A., Dertien, E.: Tinkering in informatics as teaching method. In: Proceedings of the 12th International Conference on Computer Supported Education. SCITEPRESS - Science and Technology Publications (2020). https://doi.org/10.5220/0009467304500457
12. Processing Foundation: Processing language reference (API) (2020), https://processing.org/reference/, accessed on 2020-05-27
13. Renooij, S., Witteman, C.: Talking probabilities: communicating probabilistic information with words and numbers. International Journal of Approximate Reasoning **22**(3), 169–194 (12 1999). https://doi.org/10.1016/s0888-613x(99)00027-4
14. Shiffman, D.: Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction. The Morgan Kaufmann Series in Computer Graphics, Elsevier Science (2015)
15. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. Communications of the ACM **29**(9), 850–858 (9 1986). https://doi.org/10.1145/6592.6594
16. Wilson, L.O.: The aims, goals and objectives of curriculum - what are the differences? (2014), https://thesecondprinciple.com/instructional-design/writing-curriculum/, accessed on 2020-05-07