A Modular, Practical Test for a Programming Course

Jan Stolarek
jan.stolarek@ed.ac.uk
University of Edinburgh
Edinburgh, UK
Institute of Information Technology
Lodz University of Technology
Lodz, Poland

Przemyslaw Nowak
przemyslaw.nowak@p.lodz.pl
Institute of Information Technology
Lodz University of Technology
Lodz, Poland

ABSTRACT

In order to evaluate students' programming skills during a university course, a practical programming test can be administered, in which students are required to implement a short yet complete program according to a provided specification. However, such tests often suffer from drawbacks that prevent comprehensive and accurate assessment of students' abilities. In this paper we identify these drawbacks and then present a modular, practical test that avoids common testing pitfalls, as well as show how to design such a test based on course learning outcomes. A key aspect of our approach is adoption of modularity, which ensures independent and comprehensive verification of learning outcomes. We have used our method to evaluate object-oriented programming skills of undergraduate students over several years and have found that our testing approach has proven its validity and superiority over approaches employed previously.

CCS CONCEPTS

• **Social and professional topics → Student assessment**.

KEYWORDS

practical skills testing, learning outcomes, student assessment

ACM Reference Format:
Jan Stolarek and Przemyslaw Nowak. 2020. A Modular, Practical Test for a Programming Course. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366886

1 INTRODUCTION

Verification of learning outcomes upon completion of a course is an important element of the educational process. In many courses students acquire not only theoretical knowledge, but also practical skills, and in such cases both components need to be evaluated. This is especially true for courses that teach programming as they aim not only to convey knowledge related to the underlying concepts, but also to equip students with practical abilities that allow them to develop computer programs.

Comprehensive and accurate assessment of practical programming skills is not a trivial matter. Common approaches in the literature include students being asked to read a piece of code and describe what it does or write a missing fragment of code themselves to perform a certain task [4, 5, 7, 8]. By their nature, such methods are rather fragmentary. They usually work well only when testing in isolation the understanding of fundamental programming concepts. When integration of multiple skills must be assessed, students can be required to implement a short yet complete program according to a provided specification, as was done in the famous McCracken study [9]. This is a popular approach, but in practice it is often adopted in ways that hinder accurate assessment. One common pitfall are practical tests composed of subtasks that altogether form the specification of a program, but these subtasks are not independent and build progressively one on top of another. Consequently, students who fail to complete an intermediate subtask are prevented from working on further subtasks, even if they have necessary skills to cope with them. On the other hand, the opposite can also be bad: practical tests in which requirements for lower and higher grades are completely independent from one another may lead to situations when students fulfill requirements for higher grades without satisfying requirements for lower grades, which in turn raises a question how such solutions should be graded. A common approach in these circumstances is to enforce a rule that awarding a higher grade is possible only provided that *all* requirements for lower grades are satisfied. However, the downside of this approach is that students who have acquired advanced skills but are lacking one particular basic skill necessary to complete an intermediate requirement are doomed to a low grade. Finally, some practical tests exhibit a monolithic, all-or-none structure: the specification of the program to be developed is not separated into well-defined, self-contained subtasks. In such tests it is often not clear which skills are being tested and what the exact grading criteria are. In conclusion, all those tests described above prove inadequate: they do not provide an accurate and comprehensive assessment of skills acquired during a programming course and often leave students with a feeling of being graded unfairly.

We have seen these kinds of faulty tests being administered by many of our colleagues, and also by our own teachers when we were students. Moreover, we ourselves have used such tests and were disappointed and puzzled to see that they do not provide accurate assessment of our students' programming skills. Over the years, however, we came to realize the deficiencies of our tests and came up with an alternative structure. Specifically, we developed

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366886

a modular test, whose fundamental assumption is that it should enforce demonstration of some minimum of students' knowledge and skills required to participate in follow-up courses on subsequent semesters, but then it should assess more advanced concepts and skills independently from one another. This modularity allows students who have only mastered fragments of the course material to fully demonstrate their knowledge and skills without being blocked by things they do not know. Over several years we have refined the test structure, arriving at a test that not only accurately evaluates students' programming skills, but is also considered fair by the students themselves.

2 COURSE STRUCTURE AND CONTEXT

We have developed and administered our test in the context of an "Object-Oriented Programming" course taught during the third semester of the undergraduate program in computer science. By that time students have completed a course that teaches them the basics of programming in C++ and should thus be familiar with concepts like variables, control-flow, arrays, structures, pointers, dynamic memory, and I/O.

The "Object-Oriented Programming" course lasts for 15 weeks and consists of one 90-minute lecture and one 90-minute laboratory session per week. The learning outcomes of the course are defined as follows:

(1) Name, compare, and shortly characterize different programming paradigms.
(2) Explain the idea and basic concepts of the object-oriented programming (OOP) paradigm and compare it with other programming paradigms.
(3) Discuss the basic notation of Unified Modeling Language (UML) class diagrams.
(4) Create language-independent architectural designs of object-oriented programs using UML class diagrams.
(5) List mechanisms supporting the object-oriented programming paradigm in different programming languages and discuss them in detail in the case of C++ language.
(6) Create object-oriented programs using C++ language, which are based on architectural designs prepared earlier.
(7) Use object-oriented elements of the C++ Standard Library as well as use existing and create new libraries in this language.
(8) Discuss possible variants of projects of object-oriented programs as well as possible ways of their implementation using C++ language and justify the adopted solutions.

According to Bloom's taxonomy [3] learning outcomes (1), (2), (3), and (5) fall into *knowledge* category; outcomes (1), (2), and (5) additionally include *comprehension*; outcome (6) is *application*; outcomes (4) and (7) are *synthesis*; and finally outcome (8) is *analysis*. It is worth noting that most of the above outcomes fall into lower categories of Bloom's taxonomy. This is because the course is taught in a relatively early semester (the whole undergraduate program spans seven semesters) and the level of reasoning autonomy required of students is not yet as high as it will be later. Outcomes (1)–(3) and (5) are assessed as part of a lecture exam, while outcomes (4), (6)–(8) are assessed during laboratory sessions. In the rest of this paper we restrict our attention to the laboratory sessions.

Laboratory sessions are divided into two parts. The first four weeks consist of workshops that follow the worked-out examples approach [10]. During the workshops students work in pairs to develop a small programming project designed and supervised by Teaching Assistants (TAs). The goal of workshops is to reinforce basic object-oriented concepts introduced during the lectures, give students a basic taste of the "design first, then implement" approach to software development, and teach the basics of program decomposition [6]. These skills are then further developed in the remaining weeks of the semester, when students work on their own object-oriented application, as they first prepare its architectural design, and then, once the design is accepted by a TA, proceed with the implementation.

3 A PRACTICAL LABORATORY TEST

Both the workshops and the application development described in Section 2 are done in pairs. The workshops are not graded, although students receive feedback about their solutions. The applications are graded because their development is directly linked to learning outcomes (4), (6)–(8), which require assessment. However, since students work on their applications mostly at home, it is hard to ensure that they actually work independently from other groups. Moreover, for some student pairs it proves difficult to judge each student's individual contribution. Therefore, a separate means of evaluating every student individually and under direct supervision is needed. This is attained by administering a practical test, which verifies learning outcomes (6) and (7) on an individual basis. Passing the test is mandatory to complete the course.

3.1 Test assumptions and goals

The basic assumption underlying our test is that in order to verify learning outcomes (6) and (7) students should be able to demonstrate OOP skills comparable with those acquired during the workshops and combine them with what they already learned about C++ in the previous semesters. Therefore, we expect students to at least be able to perform the following tasks:

(i) create classes and inheritance structure according to an UML class diagram;
(ii) properly initialize objects using constructors;
(iii) implement methods to read and modify fields of an object or perform other simple operations on it;
(iv) call methods of an object, both directly and via pointers – this includes calling virtual methods in derived classes via pointer to a base class;
(v) operate (create/add/remove/iterate) on basic Standard Template Library (STL) collections, like vectors or queues;
(vi) incorporate what they already know about C++ from the previous semesters (e.g., handling of streams, random number generation, use of unions, enumerations, etc.) into their object-oriented programs.

These are the basic technical requirements that every student should meet to be able to participate in courses in subsequent semesters. If a student cannot demonstrate knowledge of this material, they should not pass the test and consequently fail the course. Regarding the last bullet, it is not our intention to catch students completely off-guard by requesting them to recall something that

might have been only briefly mentioned in one of the previous semesters. We thus often narrow down the scope of the additional material they should remember from the previous semesters and inform them about it a week before the test.

We certainly wish that students have more than just the basic skills and therefore build more advanced requirements on top of the basic ones. To obtain a higher grade, students must know how to:

(vii) use static class components;
(viii) properly allocate and de-allocate memory via pointers (including virtual destructors if necessary);
(ix) extend the program with new methods not shown in an UML class diagram, based only on written description of their behavior, which requires devising proper arguments, return types, and visibility modifiers for the new methods. (In more advanced courses on programming it might be desirable to ask students to make some sort of design decisions, but we refrain from that in this introductory OOP course.)

Students take the test individually on laboratory Windows machines using the GCC compiler and the CodeBlocks integrated programming environment. During the test students are prohibited from using any automated tools that translate UML to code because that would defeat the purpose of our test. We require that before turning to automated tools students have a solid understanding of what is being automated.

An average correct solution has approximately 80–100 lines of C++ code with additional 50–80 lines in header files. A project stub with empty source files and a makefile can also be provided, if desired. The exact time limit can be adjusted to match a specific task given in the test. For us this is typically between 45 to 75 minutes. The time limit is not very generous, forcing students to demonstrate a fair amount of proficiency in developing a solution. After the time limit has passed, students are required to submit their solution for evaluation to our university's online learning platform.

3.2 The test

Below we present an example test created using our approach. In Section 4 we discuss its structure and how it fulfills requirements described in Section 3.1.

Your task is to implement a fragment of a system for managing a TODO list. Your program will store a list of tasks to do and display that list on the screen. See Figure 1 for a class diagram.

Specification:

A. *Create class structure, fields, and methods shown in Figure 1. Each class should be placed in a separate header file. Source files with .cpp extension corresponding to header files should be created in subsequent subtasks whenever necessary. (**3 points**)*

B1. *Every* `Task` *contains four fields shown in the class diagram. Implement* `get` *and* `set` *methods for all the fields except the* `id` *field, which should be initialized with an argument passed to a constructor. Once the* `id` *field has been initialized, it cannot be changed, but it can be read with the* `get` *method. We simplify the program by assuming that* `due` *and* `priority` *fields are of type* `string`*. (**1 point**)*

C. *Implement four classes for creating a string representation of each component of a task. These classes should implement the* `toString` *method inherited from the abstract base class* `Column`*. For example: the* `toString` *method in* `PriorityColumn` *class takes a* `Task` *object pointer as an argument and returns a string representation of this task's priority ("low", "normal", "high"). (**2 points**)*

D1. *Implement* `ColumnFormatter` *class for creating a string representation of a task:*

(1) *The constructor of* `ColumnFormatter` *should create an STL vector that contains instances of all four classes inheriting from the* `Column` *base class (each column should appear only once, so the collection will have four elements). (**1 point**)*
(2) *The* `taskToString` *method creates a string representation of a* `Task`*. It iterates over the* `columns` *collection to create a description of each column for a task passed as an argument. The description is returned as a single-line string. Columns should be separated with a vertical bar surrounded by spaces. (**2 points**)*

E. *Create a* `main` *function with three tasks hardcoded:*

(1) *"Take out the trash", due: 27-08-2018, priority: low*
(2) *"Return books to the library", due: 1-09-2018, priority: normal*
(3) *"Pay the bills", due: 31-08-2018, priority: high*

Place these tasks in an STL vector. Iterate over the collection of tasks and display the data of all the tasks, one task per line using the `taskToString` *method provided by* `ColumnFormatter` *class (you will need an instance of this class). (**2 points**)*

B2. (extends B1) `Task` *class contains additional* `task_no` *private static field of type* `int` *used to assign unique identifiers to tasks. The* `Task` *class constructor no longer takes* `id` *as an argument. It uses the* `task_no` *static field instead, incrementing it after each use. With this approach subsequently created tasks receive unique IDs. Initialize the static field in a way that ensures tasks will be numbered from 1. (**2 point**)*

D2. (extends D1) *The destructor of* `ColumnFormatter` *deallocates memory allocated by the constructor to objects in* `columns` *collection and clears the collection. This means that* `columns` *collection has to store pointers to objects inheriting from* `Column`*. (**2 points**)*

D3. (extends D1) *The* `Column` *class contains additional* `columnName` *abstract method for returning a string name of a given column ("ID", "Description", "Priority", "Due"). The* `ColumnFormatter` *column contains additional* `columnHeaders` *method that creates a header for the task table. The header is created by calling* `columnName` *method for subsequent elements in the* `columns` *collection, separating names of columns with a vertical bar surrounded by spaces. The resulting string is returned as the result and displayed in the* `main` *function before displaying the tasks. (**3 points**)*

3.3 Grading scale

The grading scale, as mandated by legislation in our country, comprises six grades: 2 (lowest), 3, 3.5, 4, 4.5, and 5 (highest). To pass a test a student needs to receive at least a 3. An unwritten convention at our university is that passing any test or exam requires obtaining at least 60% of all possible points. In our example test this means

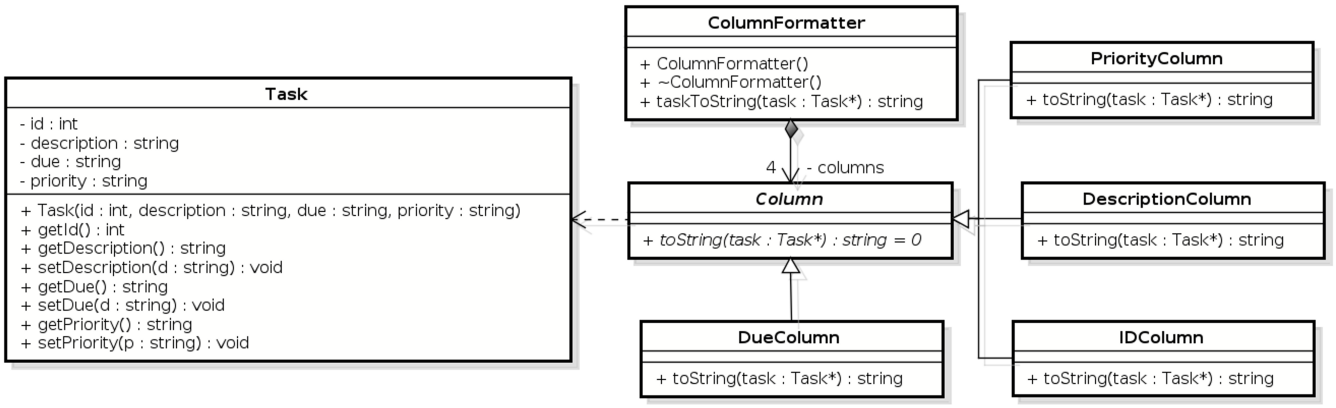

Figure 1: Class diagram for the test. Implementing some of the test requirements might necessitate slight modifications to the diagram.

scoring 11 points out of total 18 possible. Thus we grade the test using the following scale:

- 17–18 points – 5
- 16 points – 4.5
- 14–15 points – 4
- 13 points – 3.5
- 11–12 points – 3
- 0–10 points – 2 (fail)

A relatively small number of total points leaves little room for variation here. Importantly, we recognize that even the best students should be allowed to make mistakes and thus the highest grade can be obtained without accumulating all the points.

Students' solutions are graded by a TA, who first ensures that a solution compiles and, if it does, inspects the source code to verify which of the requirements have been satisfied. Each student receives a detailed breakdown of awarded points with a feedback comment where necessary. We have found that grading a solution takes around 3-4 minutes on average, so grading a group of 15 students, which is an approximate number of students assigned to a TA for a single laboratory session, should take no longer than an hour. Having multiple groups of students will of course multiply that time, which might limit the applicability of our approach if TAs are assigned too many groups. Students do not have any design freedom when implementing their solutions, so all solutions should have identical code structure and differ only in irrelevant details like loop variable names (and mistakes!). We do not rely on any automated grading tools. Solutions that do not compile receive the lowest, fail grade.

4 MODULARITY

In Section 3.1 we outlined seven basic requirements that students completing the course should meet. These are tested by the first five subtasks in our test:

- basic requirement (i) is tested by subtask **A**;
- basic requirement (ii) is tested by **B1** and **D1** bullet (1);
- basic requirement (iii) is tested by **B1**, **C**, and **D3**;
- basic requirements (iv) is tested by **D1** bullet (2), **E**, and **D3**;
- basic requirements (v) is tested by **D1** bullet (1), **E**, **D2**, and **D3**;
- basic requirements (vi) is tested by **D1** bullet (2), **E**, and **D3**.

It is crucial that all possible combinations of subtasks that grant enough points to receive at least grade 3 test all the basic requirements (i)–(vi). In particular, implementing requirements **A**, **B1**, **C**, **D1**, and **E** gives students a total of 11 points, which is exactly what suffices to pass the test. But that is not the only possible way to obtain the required number of points. A student might, for example, solve tasks **A**, **D1**, **D2**, and **D3** to obtain exactly 11 points. This is fine since these subtasks test all the requirements (i)–(vi) and so the student has proven that she has mastered the required material. Note that this makes requirement **E** optional and so it is possible for a student to pass the test by delivering a program that does not run. This might seem surprising at first but this is exactly what we wanted to achieve in our approach. All that matters is verifying the required learning outcomes, and that our test ensures.

Requirements **B2**, **D2**, and **D3** build on top of the basic ones in a modular way to test knowledge of the extended requirements:

- **B2** extends **B1**, requiring students to demonstrate knowledge of static class fields by modifying behavior of the `Task` constructor – this tests requirement (vii) from Section 3.1;
- **D2** extends **D1** by requiring students to demonstrate knowledge of C++ memory management – this tests requirement (viii);
- **D3** extends **D1**, requiring students to create new methods based on their written description – this tests requirement (ix).

As we can see, basic requirements (ii)–(vi) formulated in Section 3.1 are tested independently from one another, whereas advanced requirements (vii)–(ix) are built on top of these basic requirements, yet remain independent from one another as well. So if a student does not know how to fulfill one of the basic requirements, she can still demonstrate knowledge of the remaining ones.

4.1 Creating new tests

Once a single test is prepared following the approach described above, it can be easily altered to produce new tests that can be used with different student groups or during different academic years. For example, we have modified the example test from Section 3.2 to a version that stores a list of books in a library, with title, return date, and status ("available", "on loan") as book attributes. These two tests follow exactly the same class structure with only names of classes, fields, and methods being different. Other test topics we have used over the years include a bank/money exchange, anthill/beehive and warehouse/binder (with boxes/paper sheets being stored inside larger containers), among others.

5 VALIDITY ASSESSMENT

We now turn to addressing validity of our approach. We follow the notion of a validated assessment as introduced by Tew and Guzdial [11, 12], which in turn was adapted from the standard educational test development guidelines [1].

The conceptual content of our test is derived directly from the learning outcomes, as detailed in Section 2. This follows the assessment-driven course design approach [2]. Our test ensures that learning outcomes are being measured independently from one another, and it cannot be passed without knowledge of the outcomes deemed as mandatory. In Section 4 we analyzed this aspect theoretically and in Section 6 we verify this via an empirical study. Review of our test specification was carried out by colleague TAs teaching the same course. This ensured content validity of our test.

Per our university regulations, each student is allowed to attempt a test three times. Given that we usually taught up to two groups of students in parallel, we had to prepare up to six different tests each year. Over the three years when we carried out our research we created six different test structures, with a total of fourteen variations, as described in Section 4.1. These form our bank of test questions.

Tew and Guzdial [11, 12] aim to assess comprehension of general programming concepts and thus desire language independence of their tests. This is achieved by using a pseudocode rather than an existing programming language. This would not work in our setting because we specifically want to test knowledge of a particular programming language. We strongly believe that our approach could easily be adapted to other programming languages, but we have not yet carried out such an attempt.

We did not have pilot questions in a strict sense described in [11]. However, the act of refining our approach over the years, described in Section 6, played exactly the same role, i.e., it provided us with feedback on suitability of our questions and allowed necessary revisions.

6 PRACTICAL ANALYSIS

In this section we focus specifically on establishing construct validity [12] and reliability of our test through an empirical study. We have gathered data during three consecutive academic years. Each year approximately 160-180 students took the OOP course. For the purpose of laboratory sessions students are always divided into smaller groups counting between 12 to 16 students. We applied our testing approach in the student groups we were assigned to teach, two such groups in each of the first two years and one group in the third year, a total of 75 students.

Since the test covers material presented during the workshop sessions in the first four weeks, we initially decided to have the test in the sixth week of the course. (Not fifth, so that students could receive feedback about their fourth-week assignment and had enough time to digest it.) The assumption behind this was that it would be better to have the test while the material remains fresh in students' memory. However, we observed that too many students were struggling to even pass the test, and there were far too few higher grades than we wished. In the first year, out of 23 students taking a test after the workshop sessions, only 3 passed the test successfully, whereas 10 students delivered solutions that even did not compile. Things were equally bad during the second year, when out of 24 students again only 3 passed the test and 10 delivered non-compiling solutions. This was a clear indication that our approach of organizing the test right after the workshops was not working well. In the third year we thus decided to have the test at the end of the course, after students have completed implementation of their applications. Our conjecture was that, since the test requires practical skills rather than memorization, it would actually be better to give students more time to practice their new skills. Indeed, in the third year on the first attempt 7 students out of 15 passed the test and only 5 created non-compiling solutions. Out of 7 students that passed the test 5 received grade 4.5, one received a 5 and one a 3. This clearly shows that there was a group of students who mastered the course subject and could demonstrate it on the first attempt. This marks a significant improvement over the first two years.

All students, including those that have already passed the test, are allowed further approaches, which take place at the end of the semester. During the first year, we had a total of 22 students attempting a test for the second time: 14 of them passed the test and only 3 delivered non-compiling solutions. In the second year, 16 students took the second attempt, with 5 of them passing and 6 delivering non-compiling solutions. After the second attempt many students told us that the experience they gained during application development in the second half of the semester helped them greatly with the test, which was another indication that the test should be administered at the end of the semester. In the third year, out of 9 students 3 passed the test and 5 delivered non-compiling solutions. We have observed that out of students taking the third approach about one fourth passes the test and over a third still cannot create a solution that would compile. Although we acknowledge that with such a small sample it is hard to draw definite conclusions, the results obtained over the years show consistency, thus indicating reliability of our testing approach.

The above statistics give a broad picture of how well the students performed on a test. However, they do not answer the most important questions about construct validity of our approach:

(1) Do students take advantage of test modularity? Do they skip subtasks they do not know how to do and focus on the ones they can?
(2) Does the test enforce the course learning outcomes? Do students that pass it demonstrate the basic skills identified in Section 3.1?

To answer these questions we inspected in detail over 120 test solutions we received during these three years and carried out informal interviews with the students.

The answer to question 1 above is that indeed students use the modularity to their advantage. It is common that students skip some of the subtasks and focus on implementing ones they know how to do. Many students that receive lower grades like 3 or 3.5 do exactly that, which was our intention. However, when we first introduced this test structure, this was not always the case. To our disappointment some students would not realize that later subtasks in the test are independent from earlier ones. Interviews with students revealed that instead of skipping subtasks they did not know how to implement, students would try to implement them nevertheless and then ran out of time. This came as a very unpleasant surprise because this was exactly what we wanted to avoid! To mitigate the problem, we make it clear to students that it is correct, and in fact expected, to skip those subtasks that they cannot implement and focus on the ones they can. We include this information as part of the written instruction attached to the test and remind about it verbally about 30 minutes into the test, when many students have likely already finished implementing the first five subtasks. We have also altered numbering of subtasks. Originally we used consecutive letters of the alphabet, but then decided to number subtasks in a way that indicates dependencies between basic and extended ones. This is already shown in Section 3.2. For example, we say that **D3** builds on top of **D1** but not on any of the previous subtasks. This has brought significant improvement, although we acknowledge that this is not fully accurate: technically it is possible to implement subtask **D3** without fully implementing **D1** by creating a method for displaying column names but not for displaying the tasks. (This did not occur in any of solutions submitted by the students though.)

Regarding the question 2 above, we have found no cases where a student would pass a test without fulfilling all the basic requirements. There were several cases where students gained some points from extended requirements without satisfying all the basic ones first but none of these cases resulted in student passing a test. This is exactly what we wanted to see in our approach and also an important argument in support of validity of our approach. However, we have so far only carried out experiments in the context of a single course at one teaching institution. Further research is required to generalize our validity results beyond that context.

An important achievement of our approach is the students' feeling of being judged fairly. Interviews conducted with students show that the ability to skip too difficult requirements prevents a feeling of frustration. Students will typically comment on the requirements they did not fulfill by saying "I didn't know how to do it", understanding that their failure comes from shortcomings in their knowledge. This contrasts, for example, with results presented in [9], where students who failed the test would attribute their failure to external factors (e.g., "I didn't have enough time", "The problem was too hard").

One surprising element we have encountered from the very beginning was partial fulfillment of requirements we considered atomic. For example, in one of our tests we had an additional requirement to implement field initialization in a constructor using initializer lists in two classes and we received solutions where students would implement them in one class only. On the one hand, such solutions demonstrated that a student knew how an initializer list works, but on the other hand the requirement was only partially fulfilled. When we first encountered this situation our ad-hoc solution was to award half-points and after summing all the points round the remaining halves up in students' favor. This turned out to work well in practice, so we decided to maintain the approach of admitting half-points and incorporated them into the grading scale.

Another surprise was a high percentage of solutions that do not compile. Errors in these solutions fall into several categories. First, there are elementary C++ syntax errors, e.g., not knowing how to write a loop. Second, there are semantic errors like forgetting mandatory namespace qualifiers or not including necessary header files. These kinds of errors show that some students do not possess the skills they should have learned by completing the C++ course offered in an earlier semester. Our initial thought was that perhaps at our university testing students' knowledge was not strict enough, but we later came to realize that many other researchers had observed similar phenomena – see [5, 7–9] for examples. The third category of errors is related to new C++ object-oriented elements introduced during the workshops. A common example is omitting class name qualifier in a method definition, i.e., writing `void bar() {}` instead of `void Foo::bar() {}`. We do not yet have a method to decrease the number of these kinds of failures.

7 CONCLUSIONS

In this paper we have presented an approach to testing students' programming skills in a modular way. We have used our approach for several years and found that it works well in practice. We can now comprehensively test practical skills acquired during laboratory sessions, allowing students to independently demonstrate the pieces they have learned. Students no longer claim that the tests are somehow "unfair" because they prevent them from showing their full set of skills or focus only on a fragment of course material.

Our approach was not created instantly out of thin air. Rather it is a product of constant refinement over the years. In the beginning, we did not consciously realize the rules governing the creation of our tests and only applied them instinctively. Only with time did we realize that modularity, as described in Section 4, is the key to creating a comprehensive and reliable practical test. In this process we did not manage to avoid some pitfalls, which we described in Section 6.

In this paper we have focused on an introductory object-oriented programming course using the C++ language. Results presented in Section 6 were gathered at a single institution over the period of three years. Future work will focus on reproducing presented results in a broader context. Our first steps will be to apply our approach to a more advanced programming course with a different programming language. Also, the process of creating tests is currently based on expert knowledge and experience. In particular, assigning points for the individual subtasks to match the modularity criteria described in Section 4 might require trial and error. Developing a more precise approach to creating new tests and assigning points remains as future work. Finally, it would be an interesting experiment to test our students' mental models of object-oriented programming concepts in a fashion similar to [5] or [8].

ACKNOWLEDGMENTS

This work was supported by ERC Consolidator Grant Skye (grant number 682315).

REFERENCES

[1] American Educational Research Association, American Psychological Association, and National Council on Measurement in Education. 1999. *Standards for Educational and Psychological Testing*. American Educational Research Association.

[2] Joe Bergin, Christian Kohls, Christian Köppe, Yishay Mor, Michel Portier, Till Schümmer, and Steven Warburton. 2015. Assessment-driven course design foundational patterns. In *Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15)*. ACM, New York, NY, USA, Article 31, 13 pages. https://doi.org/10.1145/2855321.2855353

[3] Benjamin S. Bloom, Max D. Engelhart, Edward J. Furst, Walker H. Hill, and David R. Krathwohl. 1956. *Taxonomy of Educational Objectives. The Classification of Educational Goals. Handbook 1: Cognitive Domain*. Longmans Green, New York.

[4] Richard Bornat, Saeed Dehnadi, and Simon. 2008. Mental models, consistency and programming aptitude. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78 (ACE '08)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–61. http://dl.acm.org/citation.cfm?id=1379249.1379253

[5] Marilyn Ford and Sven Venema. 2010. Assessing the success of an introductory programming course. *Journal of Information Technology Education* 9 (2010), 133–145. https://doi.org/10.28945/1182

[6] Aaron Keen and Kurt Mammen. 2015. Program decomposition and complexity in CS1. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 48–53. https://doi.org/10.1145/2676723.2677219

[7] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.* 36, 4 (June 2004), 119–150. https://doi.org/10.1145/1041624.1041673

[8] Linxiao Ma, John Ferguson, Marc Roper, and Murray Wood. 2007. Investigating the viability of mental models held by novice programmers. *SIGCSE Bull.* 39, 1 (March 2007), 499–503. https://doi.org/10.1145/1227504.1227481

[9] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01)*. ACM, New York, NY, USA, 125–180. https://doi.org/10.1145/572133.572137

[10] Lai Chee Sern, Kahirol Mohd Salleh, Nor lisa Sulaiman, Mimi Mohaffyza Mohamad, and Jailani Md Yunos. 2014. The effects of worked example and problem solving on learning performance and cognitive load. In *4th Shanghai International Conference on Social Science*. 131–143.

[11] Allison Elliott Tew and Mark Guzdial. 2010. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10)*. ACM, New York, NY, USA, 97–101. https://doi.org/10.1145/1734263.1734297

[12] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*. ACM, New York, NY, USA, 111–116. https://doi.org/10.1145/1953163.1953200