INTRODUCTORY IT COURSES WITH LARGE CLASSES: PERFECT SETTING FOR A PSI REVIVAL?

Guttorm Sindre
Department of Computer Science (IDI)
Norwegian University of Science and Technology (NTNU)
guttorm.sindre at ntnu.no, +47 7359 4479

Abstract: *The Personalized System of Instruction (PSI), a mastery learning approach, attracted a lot of interest in the 1970's. However, in spite of largely positive results in empirical studies of learning effect and student satisfaction, PSI gradually lost ground through the 80's and 90's and is now unknown to many educators. A 2007 article by Eyre asks directly in its title "Was it a Fleeting Fancy or is there a Revival on the Horizon?" – pointing to improved e-learning technology as a way to mitigate the high work burden PSI might otherwise cause on the teaching staff.*

This paper makes a more concrete discussion of the prospects of such revival, considering if introductory IT and programming courses might be a perfect setting for a modern approach to PSI, and especially if it would be suitable for large classes. A possible argument against this is that the big class sizes that such courses tend to have in Norwegian universities would discourage experimentation with major changes of teaching strategy. On the other hand, there are also many factors in favour of PSI or similar mastery learning strategies in introductory IT courses. The paper analyses potential pros and cons of going PSI in an introductory IT course, and concludes with a concrete proposal for how such a transition might be implemented within the limits of typical current restrictions for courses and degree programs.

Keywords: mastery learning, PSI, information technology, programming

1. INTRODUCTION

The Personalized System of Instruction (PSI), also called the Keller Plan, was introduced in the 1960’s by Fred Keller and colleagues, while teaching Psychology at the University of Brasilia (Keller 1967, Keller 1968). The five key principles of the original PSI were:

- *Units of content:* A course should be decomposed into a meaningful sequence of units.
- *Unit mastery:* passing a preceding unit would be a prerequisite for continuing to the next one. A pass would require a high threshold such as a score of at least 90% on the test for that unit. A failing student would have to keep on studying the same unit and try the test again later.
- *Student self-pacing*: Each student could choose a personal pace of progress. Hence, stronger students might complete each unit, and the course as a whole, more quickly, while weaker students might spend more time.
- *Written materials* should be used to present new content (rather than, e.g., lectures).
- *Tutors* had a central role, both in correcting tests and giving instruction and *frequent feedback* to students about their progress.

The preference for written material was inspired by findings that lectures had limited learning effect, and for better supporting self-pacing: Written material can be read at any time and pace the student prefers, while a lecture series tries to force the student to adapt to one specific pace through the curriculum. Some may think that by abandoning lectures, PSI would abandon the triple sequence of (i) presentation of principles, (ii) practice, (iii) comparing practice with principles, which has been found effective for teaching cognitive skills (Mayer 1989, Gagne, Briggs et al. 1992). This is however not the case, as *presentation* does not have to mean lectures – principles may equally well be accessed by reading a textbook or a PSI study guide written by the teacher. Nowadays, the material need not be text on paper. In a revision of the key features of PSI, Fox (2004) generalizes to *on-demand course content,* which may include instructional videos, interactive textbooks with self-test quizzes, etc. Moreover, Fox presents *immediate feedback* as a principle in its own right, not necessarily part of the tutoring, since feedback could be partly automated.

Concerning learning outcomes and student satisfaction, PSI has had a lot of promising empirical results compared to more traditional teaching strategies (Kulik, Kulik et al. 1979, Fraser, Walberg et al. 1987). There were also some concerns, a major one being student procrastination (Reiser 1984). Caldwell (1985) also raises two other concerns, namely cheating by tutors becoming too friendly with students, thus letting them passing tests too easily, and unit tests biased towards low level learning outcomes. In spite of such concerns, many saw more advantages than disadvantages, and PSI had an enthusiastic following with a strong belief that its usage would increase to challenge the dominant “same pace, different learning” paradigm. However, this did not happen. From the mid 80’s onwards, interest in PSI gradually declined, so that the recent annual amount of research papers reporting on trials of PSI is much smaller than it was in the 70’s. Several reasons have been suggested for the decline of PSI (Buskist, Cush et al. 1991, Sherman 1992): The discouragement of lectures may have been a too sharp deviation from the normal practice at universities. Student self-pacing was at odds with university administrative routines and typical semester and degree program structures. PSI’s need for frequent testing, for which the teacher had to develop study questions and test items, put a higher burden on teaching staff than traditional teaching with a lecture series followed by an end-of-course exam.

The decline of PSI does not mean that it is entirely gone, and improvements in e-learning technology could alleviate some of the problems that worked against PSI in the past. Already in the early 80’s there were proposals to ease the burden of student tutors by automating multiple choice tests (Crowell, Quintanar et al. 1981). Later the term *computer-aided personalized system of instruction* (CAPSI) was used for approaches including a focus on peer review (Pear and Kinsner 1988), to allow for effective mass-evaluation also of essay-type questions. Eyre (2007) asks whether there is a PSI revival on the horizon, observing that many educators were making use of e-assessment technology to achieve the frequent testing that was previously very labor intensive. For instance, CAPSI was found to reduce dropout in blended learning courses in the study (Svenningsen 2009). An ICT-supported application of PSI also gave good results in an introductory discrete mathematics course for CS students (Rae and Samuels 2011), concluding that “this pedagogy is promoted as an effective approach to teaching in higher education, especially the teaching of cognitive skills to diverse cohorts of students on foundation level modules.” (p.2423). Other recent empirical investigations of PSI and similar mastery-learning approaches have also shown promising results, e.g. (Foss, Foss et al. 2014, Lacroix, McCall III et al. 2014, Fleming, Barth et al. 2018).

Most interestingly, PSI has been used since 2011 in an introductory programming course at the University of Agder, Norway. The experiences from the first five years are reported by (Purao, Sein et al. 2017). While the initial direct implementation of Keller’s original principles had considerable challenges – but so had the previous more traditional course design prior to 2011. With some adaptations developed over the next years, mainly to mitigate procrastination, the approach is evaluated as quite successful. Currently they have two courses on OO programming with Java using a PSI approach, namely IS-109 (5 ECTS credits) in the first semester, and then the follow-up IS-110 (10 ECTS credits) in the second semester. Both courses use the same textbook (Barnes and Kölling 2016), with IS-109 covering chapters 1-5 and IS-110 chapters 6-15. Both courses are divided into modules corresponding to book chapters. IS-109 is assessed pass/fail on a portfolio of 5 module assignments which each student can do in a preferred pace, but all modules eventually have to be passed to pass the course. For IS-110, a student has to complete a minimum 5 of its 10 modules to be entitled to sit an end-of-semester oral exam (15 minutes per student) which determines the grade. For a student who has only done 5 modules of IS-110, the best possible grade to achieve for the oral exam will be C.

The courses at UiA had 60-80 students per year during the reported period (currently around 80 passing the oral exam), while at the NTNU the early programming courses are much bigger, for instance, TDT4110 Introduction to IT (first semester, Python): 1850 students, TDT4100 OO Programming (second semester, Java): 700 students, TDT4102 Procedural and OO Programming (second semester, C++): 750 students. Hence, the problem of scaling is more complicated, for instance meaning that oral exams, even with a short duration of 15 minutes per student, would likely not work. The questions posed in this paper are: *1) Could PSI be suitable for introductory IT courses, for instance university level beginners' courses in programming – and even for large classes? 2) What might a PSI implementation of such a course look like, assuming that it would still be required to comply with the typical course structure and administrative routines dominating Norwegian universities today?* An issue for the second question would be the typical requirements that a have a fixed size and duration (e.g., one semester, 7.5 credits at the NTNU). This might not go well with the PSI principle of student self-pacing, where a stronger student might complete the course in half a semester while a weaker student might need 1.5-2 semesters.

We acknowledge at once that this paper does *not* present any empirical findings of its own. There is, as of yet, no trial of PSI to be reported on from an introductory programming course in the author's institution. Nor are there much empirical results from elsewhere, except for the experiences from the University of Agder (Purao, Sein et al. 2017), which were with classes an order of magnitude smaller than what we would be facing. Hence, it is more of a *vision* paper, discussing how PSI might be used in large class introductory programming courses, based on literature about PSI and the author's knowledge of typical learning goals and contents of such courses. The justification for the paper is that a switch to PSI would be such a radical deviation from current dominant practice that it would be a high risk endeavor in a first semester course with many students – yet also something that could be highly beneficial if successful, given PSI's track record of positive results in the past. Hence, it would be an interesting topic for discussion among teachers even beforehand: Is this something worth considering or not?

The rest of the paper is structured as follows: Section 2 presents argument for and against a PSI approach to introductory IT courses. Section 3 presents a proposed implementation. Section 4 then discusses potential benefits and challenges with the proposed implementation, as well as limitations with the paper. Section 5 concludes the paper.

2. ARGUMENTS FOR AND AGAINST PSI

A general argument for PSI is that *one-pace-fits-all*, which is the underlying assumption or necessity of a traditionally run class, will be inadequate to students at both ends of the spectrum. For instance, a lecture series necessarily follows a certain pace through the curriculum. The lecturer may adjust the pace if there is a feeling that most students think it is too fast, or too slow, but *some* choice of pace must necessarily be made. This means that (a) weak students who need more time, will experience that the lecturer moves on to the next subject before they have understood the previous one, or (b) strong students will experience that the teacher is tea-spooning and repeating content that they have already understood. Normally, both (a) and

(b) will happen to some extent, as the pace is set to cater to a majority of mid-range students. Similarly, weekly compulsory exercises used in many courses will assume a certain pace of progress, where some students could have gone faster and others might have needed to go slower. For the weaker group, need to deliver a compulsory exercise on subject matter not yet understood may lead to copying, or delivery of barely passable work based on limited understanding of the subject – in both cases giving limited learning effect from the exercises. Especially in courses where next week's subject builds on the previous one, learning of the next subject will also be hindered by the lack of foundational knowledge, and the weak student will be constantly stressed to catch up but never getting the time to fully do so.

The above argument could apply to almost any course in any discipline. However, introductory IT and programming courses often have the following properties:

High variation in students' ability and previous knowledge. While this can be true for many introductory courses at universities, other courses – such as math and science-related subjects – can at least base themselves on the assumption that students have had some common, compulsory courses in secondary education. In many degree programs students will have been selected based on good grades in such courses, so that substantial previous knowledge could be assumed. As for programming or IT, there are no dedicated compulsory courses in Norway's secondary education. Hence classes may easily have the full spectrum from students who never wrote a single line of code, and also made limited use of PC's in other respects, to students who have been programming for years, as a hobby and/or through elective courses in secondary education. The students who know a lot from before could have gone much faster. Some might even be capable of an A already at the start of the semester if offered an exam to prove it. However, the teacher must adapt to the "least common divisor" of no previous knowledge.

A strong tendency for subsequent parts of a course to build on the early parts. For instance, in an introductory programming course, basic datatypes, variables, loops and branching structures, function calls etc., will continue to be used also when more advanced topics are covered. A student who has not yet understood such early concepts, will continue to struggle through the rest of the course – understanding new concepts is even harder when previous ones that they build on have not yet been properly understood. Of course, there are similar challenges in many other disciplines – but not always to the same extent. While programming is about making something from smaller building blocks, many other courses have a more encyclopedic nature where parts build on each other to a lesser extent.

In programming, broad but low mastery has lower utility than narrow, high mastery. In many other disciplines, it may be less clear what is best: to have a wide but shallow understanding, or a narrower and deeper understanding? Is it more useful to know a little bit about many philosophers, many historical periods, or many physical phenomena – or to have a deep understanding of just a few philosophers, historical periods, or physical phenomena? In coding the choice seems more obvious. It must be better to be able to write code that works with a limited subset of a programming language, than to have vague knowledge of many different programming mechanisms yet being unable to write working code with any of them.

Automated assessment without losing constructive alignment. As mentioned in the introduction, one criticism against PSI – or at least some implementations of PSI – was that module tests tended to focus on low level learning outcomes (Caldwell 1985). The need for frequent tests in PSI – maybe every week – would easily overburden teaching staff with correction tasks (traditionally done by tutors), unless the tests were really quick to correct. This would result in a preference for question genres like multiple choice and fill-in-blank. Clearly, such questions would be insufficient for assessing whether a candidate is able to program, and the principle of constructive alignment indicates that assessment tasks and learning activities should be closely aligned with each other, and with the learning goals of the course. An advantage that programming has over many other disciplines, however, is that the typical outcome also of more complex problem-solving and creative tasks would be something that can be automatically assessed, namely program code. It can be run against a test suite to see if it actually does what it is supposed to, as well as analyzed structurally to see if it complies with coding standards.

Arguments against PSI in an introductory programming course might be partly practical, partly principal. Practically, it might be problematic to run one course in PSI fashion, alongside other courses that follow a traditional one-pace-fits-all style. One-pace-fits-all also has clear advantages concerning low cost (e.g., offering the same lecture series and same progression of weekly exercises to the entire class), so attempts at PSI could run the risk of either being very expensive, or alternatively, if sticking to a limited budget, being unable to really implement PSI principles and offer each student the supposed tutoring. If there is a course the next semester which builds on the PSI course and slow pace students are not yet finished with the PSI course when the next semester starts, these will lack e.g. the latter half of the curriculum of the PSI course, which the next course maybe wants to build on.

A principal objection could be related to the social aspect. Even if lectures may have limited learning effect, they also have a social function, giving students a reason to go to campus. In lectures, they meet their classmates, and to some extent the teacher (though the teacher-student encounter is asymmetric and impersonal for most, when there are 3-400 students in the lecture theater). PSI – especially if implemented with fully automated testing – essentially sees the student pursuing the learning alone, at a personal preferred pace. Especially in a first semester course, when some students have few friends or acquaintances, the teaching should also help them acquire a social network. Team-based learning (Michaelsen and Sweet 2011) as successfully used in Norway for instance by (Kraemer 2017), would help the students getting acquainted with some of their classmates, but does not fit PSI's idea of self-pacing, since it relies on the class progressing jointly through a series of readiness tests. Pair programming is another technique that has been successfully in CS1 courses (Williams, Wiebe et al. 2002), again, self-pacing might be a problem if forming student pairs at the outset to last throughout the course. However, recommendations are anyway that student pairs should be compatible in skills (Salleh, Mendes et al. 2011). Since a first semester course lacks previous knowledge about programming skills, it might anyway be preferable to get started with some tests, and then after a couple of weeks pair students who are going at the same pace. With many hundred students taking a course in total, there are likely multiple students at any pace and ambition level, thus possible to find a suitable peer for everybody.

A more specific, programming-related argument against PSI relying on fully automated tests is that even if program code *can* be assessed by automated tests, this would limit the assessment to fairly short programming tasks with an already known solution. However, an important task for an introductory course in programming could also be to create curiosity and enthusiasm about programming, thus allowing for some more creative tasks. Moreover, many first year students have moved away from home to study and have few close friends at the outset. If a course leaves each student on their own PC, learning through an automated test application with pointers to other digital resources, this will not help the students to get a social network.

In the next section, we will present a possible implementation of PSI and discuss whether it alleviates these problems.

3. OUTLINE FOR A PSI APPROACH TO NOVICE PROGRAMMING

A central element of PSI is that the student must master the current unit before moving on to the next. A key decision for designing a course according to PSI principles is therefore how it can be decomposed into a sequence of units. Literature recommends against putting too much into material into each unit, as units should appear manageable to students. This points in the direction of having many small units rather than few big ones. For instance, Lewis and Wolf (1973), reporting on the usage of PSI in a first semester chemistry course, and then in a second semester chemistry course, had these courses designed with 18 units, where normal progress would be to do one each week. In addition to units with tests, students also had to do laboratory work. The grade was based on the number of units the students were able to pass, while the lab work was just pass/fail.

Similar to the approach of (Lewis and Wolf 1973) with a course combining a test ladder with laboratory work, the suggestion here is to have an introductory programming course combining a test ladder with a programming project. Both the test ladder and the project could be worked on during the entire length of the semester, though students would typically need to pass some mastery tests before getting up to speed with their project. A possible 14 unit breakdown of a course in procedural programming in Python is shown in Table 1, the column "Topic" indicating what topics are introduced in the various units. Concepts introduced early, like variables, branching, loops, and functions, will keep on being used throughout the later units, too. For instance, it is hard to make much of programming with lists without using loops, and for programs of any length, it makes sense to divide into several functions. Hence, the early units do not have to include everything about concepts introduced there, which might be beyond weaker students at that point.

Table 1: Possible layout of a Python course with 14 units

Unit	Topic	Grade	LOC
1	Screen output, keyboard input, editor usage		
2	Turtle graphics		
3	Variables, assignment, datatypes, arithmetic expressions		
4	Branching and logical conditions		
5	Loops		
6	Functions, introduction	E	100
7	Functions (cont.), modules and import		
8	String processing	D	150
9	Lists		
10	Text files	C	200
11	Sets, dictionaries, binary files		
12	Exception handling	B	250
13	Recursion		
14	Advanced problems combining everything	A	300

In line with PSI principles, a high threshold of mastery should be required for each unit, for instance scoring at least 90% on the corresponding test, after which the student can begin working on the next unit. The column "Grade" indicates the link between units and grades. As can be seen, units 1-5 are just stepping stones towards the vital "E-test" for unit 6. Similarly, the odd numbered units further down are also just intermediate between two grades. Why these intermediate tests – rather than a much simpler schemes where all tests have grade impact (e.g., just 5 units, 5 tests, the E-test, D-test, C-test, B-test, and A-test)? The point is that this would yield way too big units. One PSI guideline is that units should be of manageable size, and that especially the first units should be simple to give students confidence from an early success. For instance, Lewis and Wolf (1973) used 18 units for a first semester Chemistry course, and decided to continue with 18 units when they took PSI further also to a second semester course. With just 5 units, our course would have several different topics into the same unit, and especially the E-unit would be quite big and unmanageable. The weakest students would be struggling with that single unit the entire semester, with repeated failures at tests, thus getting little sense of progress during the semester.

The combination of a test ladder and project work means that passing unit 6 would not immediately yield an E, the student also has to deliver a programming project. The purpose of this project is to support some objectives not easily covered by the testing ladder:

- Learning to write longer programs decomposed into various functions. This is not easily evaluated just by automated tests of short duration.
- Stimulating curiosity, joy, and creativity of coding. Creativity is hardly stimulated by just solving set problems made by a teacher. As indicated by (Lauvås and Raaen 2017), employers are looking for programmers who are passionate about coding – hence universities should try to encourage this passion from the start.
- Stimulating collaboration, social inclusion and belonging. Having the project done in a team context could guarantee that every student gets to know quite well at least 3-4 classmates during the course, thus combating freshman loneliness. This could be done such that the team develops one joint programming project, but in case this is difficult due to huge variation in student pace, they might instead have joint meetings with a common TA, presenting their projects so far and getting comments from other students in the group.

The column "LOC" indicates required length of the program that a student has to write to achieve a certain grade for a project, assuming all these lines are meaningful (i.e., not comments or dummy statements). Moreover, the program has to make use of the concepts introduced that far. For E the program must thus include user i/o, variables, assignment, arithmetic expressions, branching, loops and some simple self-defined functions. For D it must add functions with some more advanced features (e.g., return values, default arguments, local variables) as well as string processing. If several students make a joint software product, comments must show who has written what. This would enable students at different levels of ambition to collaborate on the same product without getting into conflict; the E student simply has to write 100 lines of the code, demonstrating use of the most basic programming mechanisms, while the A student would have to write 300 lines and demonstrate a much wider repertoire.

Apart from length and demand to use certain concepts, as well as running without failure, there are *no requirements* put on the programming project. Each student can program whatever (s)he thinks is fun or cool (as long as it does not transgress legal or ethical boundaries, e.g., making a virus or a program that displays pornographic pictures should not be welcomed). The motivation for this is as follows:

- The project is primarily meant to be fun and motivating, not difficult. The test ladder is the main gatekeeper with respect to mastery, and someone who has passed the unit test for a certain grade, should easily be capable of delivering a project for that grade, too.
- While the supervised tests are hard to cheat at, the project – which the student must be able to work on unsupervised – would be easy to cheat at, for instance having somebody else write much of the code for you. Hence, if the project was made difficult – so that in many cases it would be project quality rather than test passage that held somebody back from a better grade, the course would be likely to develop a serious cheating problem.
- Detailed evaluation of projects for quality would be very time-consuming, and even more so if also needing an effort to mitigate cheating on the project.

A necessary implication of the above is that the grade for the course will be the **minimum** of the test ladder grade and project grade, not the average – otherwise it would be way too easy to get a good grade by cheating on the project. I.e., if somebody delivers a project worth A but only manages to pass the E-test (i.e., unit 6), the resulting grade is E, not C. For an opposite case, if a student passes tests all the way to A, but only delivers a project with 50 LOC, the grade will be F. Presumably, this will be rare – a student able to pass all the tests should easily be capable of writing more code, especially when there are no special demands to the type of application.

With a scheme like that of Table 1, flexible pacing could be achieved by letting each individual student progress through units and passing tests at their own preferred pace, i.e., a student could show up for a test when ready for it, rather than at specific dates set by the teacher. Each test need not last long, a typical maximum duration could be 1 hour. This would yield 6 hours of testing for an E, 14 hours for an A, which allows much better coverage of the subject than the current 4 hours for a single end-of-course exam. Of course, assuming that tests must be done under supervision to mitigate cheating, the test facility cannot be open 24x7, so a student cannot demand to be tested at any The quickest students – typically those who are skilled at programming from before – might be able to do all the tests within the first couple of weeks. A clever and ambitious student without previous knowledge of programming might instead reach the A by going at even progress throughout the semester, passing tests approximately once a week. A student who is content with C, or for whom C is the realistic level of achievement, might go a little slower than the A student, for instance passing C with two weeks to go. The student might then try for B but not quite make it, or decide not to try for B – for instance due to anxiety for another course taken in parallel (e.g., math) and thus prioritizing to spend extra time on that instead. A weak student who really struggles with the course might be able to just pass the 6 necessary units yielding E, for instance at study weeks 2, 4, 6, 8, 11, 15, respectively.

To help students know whether they are ready for a test, it would be a great advantage if they could self-test (at home or anywhere) against the same question bank as used for the supervised tests. If a student sees that twice in a row, she does 95% on the D-test, she can then conclude: Yes, let’s go and take it tomorrow! On the other hand, if the score is around 75%, the student is likely not ready and need to study more, then do another self-test to see improvement. Self-testing would be the course’s main approach to *immediate feedback,* which (Fox 2004) emphasizes as a key property of a modern PSI course. Another key property is *on-demand course content*. For a programming course, this could be wikis, textbooks (on paper, as well as e-books, either static or interactive), instructional videos, etc. Typically, self-test results should guide students towards available resources. For instance, the student who got just 75% on a self-test for unit 8 (string processing) could be informed that most of the lacking points were lost on questions relating to string slicing and finding substrings within strings, with pointers to learning resources explaining these topics.

In Keller’s original application of PSI, tests were corrected manually by tutors (also called proctors), similar to teaching assistants, who then gave the students feedback on their progress. As pointed out by e.g. (Fox 2004, Eyre 2007) it might be more efficient to have tests automatically scored, which would also be our proposal for a programming course. Test items could include several different genres, such as multiple choice, short answer (e.g., program comprehension tasks), fill-in-blank (e.g., add one missing line to an almost complete program to make it work), drag-and-drop, and programming[1]. The genre drag-and-drop can be used to create so-called Parsons’ problems (Parsons and Haden 2006, Denny, Luxton-Reilly et al. 2008, Helminen, Ihantola et al. 2012), which are easy to score automatically yet have been shown to correlate well with traditional programming questions, and to have good learning effect for programming skills.

Freed from the traditional task of correcting tests, the tutors – in our case Teaching Assistants – could instead spend more time on face-to-face interaction with students. Typical tasks could be to give feedback on their project code, and to help them in cases where they do not understand the on-demand learning resources and need extra explanation or advice.

[1] Currently, Inspera (the system used for digital exams at the NTNU and several other Norwegian universities) only offers programming tasks as a manually graded question genre. However, they are working on Beta functionality where students can compile and run the code in the web browser during the exam, and where code can be evaluated against a test suite – which would allow for automated grading also of such tasks. It is reasonable to assume that this functionality will be generally available long before any intro programming course is switched to PSI.

In an ideal PSI world, a student who has not come all the way to A during the semester, should keep on with the course also in the next semester. If the introductory programming course in the Autumn has a kind of continuation in a more advanced programming course the next Spring, a student who has – say – reached C in the intro course in December, should keep going through January and February to reach A, and only then start with the more advanced programming course. However, this might clash too much with the current way of doing things. The advanced course might not be run according to PSI and is anyway supposed to start in January. The administration might demand that grades for the intro course are reported in December / January, based on achievements by the end of the semester. Hence, we assume that issues surrounding grading must be handled in the following way:

- The student gets the grade that was achieved by a certain deadline at the end of semester (i.e., minimum of test ladder grade and project grade)
- Students who failed due to not reaching high enough on the test ladder, can retest in the next semester. This would be similar to a continuation exam for an ordinary end-of-course exam, though it would not make sense to demand that the student wait all the way until July/August to retest for a December failure. Rather, the test when ready principle suggests that the student could work on, during the Xmas break, or during January, do some self-testing to check for readiness, and then show up for tests. Of course, it might not be feasible to have extensive opening hours for testing in the opposite semester, but having enough to cater for those who failed would presumably be good for throughput and retention. The failed student should continue from the first test not passed, i.e. a student who only passed units 1-3 during the semester, would have to pass 4, then 5, then 6 for the "continuation".
- Students who failed due to a lacking project (probably rare) would have to extend and then demo their project for the "continuation" – again allowed any time you like. For both kinds of continuation, of course, staff might collect results from many students and then delay the reporting of them until the continuation season if – for some reason – administrative procedures will not accept to receive the results earlier.
- Students who have passed but want to improve their grade – similar to a voluntary retake of an end-of-course exam – can also start from where they left off. Hence, a student who passed up to and including test 10 and got a C, would subsequently need to pass test 11 and 12 to improve this to a B, plus extend the project with 50 lines applying sets, dictionaries, binary files and exception handling.

A possible advantage of this more flexible scheme is that they can pursue the previously failed tests at the beginning of the next semester, when everything is still fresh in memory. For many, this would be better than waiting half a year for the continuation exam (in case of F) or a full year for a voluntary retake (in case of a poor passing grade). Of course, the waiting time may also be an advantage for some, e.g., gaining maturity, so the main point is that the student is free to choose: Want to wait until summer and they retry? Fine. Want to retry in January? Also fine. Automated tests of limited cost are key for being able to allow this. In the opposite semester, supervised testing might only be available a couple of hours per month, to avoid prohibitive costs for invigilator person-hours.

4. DISCUSSION

4.1 Possible advantages

A scheme like the one outlined above could have several potential advantages for students:

- Frequent and immediate feedback on your progress, through supervised tests that may typically be taken once a week, and even more frequent through self-testing that can be done as often as you like.
- Compared to courses that rely just on a final exam for the grade: reduced risk for the student. With the entire grade given on basis of one exam, your performance on that particular day becomes very decisive – and potentially vulnerable. Spreading the data collection for grading across many smaller tests reduces this risk.
- Compared to courses that have several tests during the semester that average to a grade: ensuring the same grade for the same end-of-semester mastery. For the averaging approach, this may not be the case. Assume a course which has three tests each counting 1/3 of the grade, at week 5, 10, and 15, and three students X, Y, Z, who all reach an 85% mastery of the intended learning outcomes by the end-of-semester. They should thus do equally well on the third test, but not necessarily on the first and second, since they may have reached their 85% mastery in different ways. A slow starter, X gets D's on the early tests, averaging to C. A steady pace student, Y performs consistently at B. Knowing programming from high school, Z gets A's on the first two tests – but does nothing during the semester so ends at the same 85% mastery as X and Y – but averages to A. Thus, three students with the same end competence, but three different grades. With PSI, the grade defined by the number of units mastered, all would get the same grade, more correctly reflecting final competence.
- Especially for the weakest students, making it extremely clear what they need to learn: Enough to pass the E-test. The scheme also allows them to keep on concentrating on immediate obstacles for progress, rather than forcing them to jump to the next topic without having understood the previous one.
- For strong students, being allowed to finish the course with an A early in the semester, rather than being held back by a one-pace fits all schedule with boring lectures about topics they already have detailed knowledge of. This can be utilized in several ways: use the freed time to work harder in other courses, or voluntarily choose to learn something more, for curiosity or increasing their potential to qualify for relevant summer jobs.
- For students across the spectrum, a bigger sense of control over their learning, being able to go at the pace that fits them best, setting their own ambition level, and getting precise feedback whether your progress is according to ambition, an advantage for self-efficacy.

Apart from the above pedagogical advantages, one might also hope that grade complaints will be fewer if tests are fully automated and with clear rules that 90% are required for a pass. Of course, there is also the project, but the evaluation of the project is not very strict – just a count of lines, a quick check that the required concepts / mechanisms have been used, and being able to demo for the TA that the code is running. Hence it will seldom be the project that is holding somebody back from getting a better grade.

4.2 Possible disadvantages

There are also possible disadvantages of the proposed course design. PSI literature in general admits that the work burden for teaching staff can be high compared to traditional lecture courses. Although automated testing can alleviate some of this, these tests and the corresponding learning resources first have to be developed.

Huge up-front investment. If the question base used for all the tests is to be available also for student self-testing – which would be a big advantage pedagogically – it has to be huge. Not just big enough that students do not get the same questions over again when retaking failed tests (which might lead them to just repeated tries without learning in between), but also big enough that it does not matter if students start a joint effort of assembling lists of known questions and publishing these with correct answers on social media. I.e., there must be so many questions that it is unrealistic for anybody to memorize questions and answers rather than having to acquire the necessary knowledge and skills to pass a test. It would then be insufficient to have 100 items for each unit, probably it should rather be in the order of 1000 – meaning that the total test base covering all units (i)-(xii) as outlined in section 3 would have to consist of tens of thousands of question items. Developing all of these will be a considerable investment. However, the cost and time needed for this could be reduced in several ways:

- *Auto-generating items*. Many types of questions could be generated in huge numbers just by making a question template and then defining a range of possible variations over this template. For instance, code comprehension items (e.g., "What will be printed by this program?") can start from a template allowing a number of variations that can include different values for variables, altering arithmetic operators between +, -, /, * and relationship operators between <, >, <=, >=, changing intervals and step increment values for loops, changing the order of some sentences, etc. By combinatorial explosion, this could easily twenty different items from the same question template, in some cases much more. A similar approach could be used for fill-in-blank questions, auto-generating several different programs from one template, including in the question statement what the wanted result is, and then remove one code line (which the student must then write). Also for traditional programming questions, such a variations over a theme approach could be effective. For instance, with the typical problem of iterating through a list to either find or aggregate some result, the problem could be to find whether a specific value is in the list, or to find the biggest / smallest / second biggest / …, or to find some pattern where a certain value comes just before or after some other value. Independently of this the wanted answer might be the target value's position(s) in the list, the number of occurrences; etc. Thus, from one template a wide variety of list iteration programming tasks could be generated.
- *Crowdsourcing.* For questions that cannot be generated automatically, crowdsourcing could be an option. Teachers all over the world could share the work burden of creating a huge question base with various types of items for popular programming languages (e.g., multiple choice, short answer, fill-in-blank, drag-and-drop, programming…) which all could later benefit from. This could also give an additional possibility for sharing test data and experiences, so that teachers could see how their students are doing relative to others. Another option is crowdsourcing to the students the task of proposing test questions. Proposals must of course be quality assured by a teacher before being entered into the question base. However, with 2000 students per year, proposing – say – 3 question items each, even if only 10% of the questions end up being used, that is still 600 new questions added to the bank.
- *Buying questions from international test banks?* Not free, but presumably much cheaper than having teachers develop them manually – although questions would have to be translated to Norwegian.

Even with measures such as the above, there is undoubtedly a considerable investment needed. However, the hope would be that in subsequent years, costs would be lower.

High operating cost of the testing scheme? Even if tests are computerized and auto-scored, so that self-tests can be done at home and be rather cheap, the formal tests must be done at school under supervision of invigilators to mitigate cheating – otherwise it would be too easy for weak students to have stronger students do the tests for them. This means that testing will have both an area cost (a room of sufficient size), and a personnel cost (invigilators). A room seating 50 students, open for 60 hours per week (5 days x 12 hours 08-20, would suffice for a course of 2000 students if each student takes on average 1.5 tests per week. Likely, there would have to be a kind of booking scheme, students having priority for various time slots that avoid collisions with learning activities in other parallel courses. To avoid conflicts with other courses, students should not be allowed to test at times that collide with other organized learning activities for their degree program. Moreover, students should be discouraged from sitting formal tests prematurely, and from retesting without any intermediate learning just in the hope of getting lucky. Hence, it might be feasible to have a retake quarantine after a failed test, for instance, 3 days if the student was close, a full week if far away from the target. A booking scheme would also make it easier to know the number of invigilators needed somewhat in advance, so that you avoid hiring too many invigilators only to find that few students show up for testing. The fact that students testing at the same time do not get the same questions – but have questions drawn randomly from a much larger question bank – does mitigate some cheating risks, and some other cheating risks can be mitigated by digital exam technology. However, a modern approach to cheating, like using communication equipment and small wireless earpieces to get help from more competent outsiders during the exam, is still a considerable threat. A scheme where students need at least 6 hours of testing for E, 14 hours for A, and where they can be rather flexible about when to take tests, will probably be substantially more costly than one where every student just goes through a 4 hour test at a set date at the end of the semester.

Frequent testing stressful? Especially for students with test anxiety, the frequent testing may also be stressful, like doing tests every week rather than just one at the end-of-semester, or maybe just two midterms counting 25% each, and then a final exam counting 50%. However, compared to the exam-only case, there are also some possible advantages. With exam-only, the entire grade depends on your performance on that single day. Students with test anxiety may feel less stressed early in the semester because that day is still far into the future, but will feel worse when it is getting closer. A good thing about our scheme is that every success is retained, whereas every failure can be improved upon. If you pass the E-test, other test failures further into the semester will never take that E away from you, while if you fail the E-test, you can try again the next week, and do a lot of self-tests and additional learning effort to improve on the type of questions you got wrong in the meantime.

Procrastination? As reported by Purao et al. (2017), most of their adaptations from their initial implementation of PSI in 2011 towards the currently used approach, had the purpose of addressing procrastination. The outlined scheme offers some carrots which may work against procrastination: Unlike other courses, it is possible to secure a passing grade quite early in the semester, if you put in effort from the very start. Many freshmen feel a lack of control at university compared to high school, where they had frequent graded tests and thus a better idea of where they stood in various courses. In a PSI course they could regain some of that control if they put in early effort, like securing an E already in the first third of the semester. Hence, it is not obvious that students will respond with more procrastination. Some who have poor self-discipline may be likely to procrastinate, while others may decide to put in more effort early in the semester with the given scheme. To help those who would otherwise procrastinate, it might be an idea to have some minimal progress requirements. For instance, that units 3, 4 and 5 in Table 1 must be passed with at least 9, 6, and 3 weeks to go of the semester, respectively, thus guaranteeing that every candidate has a decent shot at an E. Similarly, there might be requirements for the project, having at leat 25, 50, and 75 lines of code by the same deadlines, to avoid relying on coding the whole thing at the very end of the semester. For all students, regardless of progress, it might be compulsory to show up for a 1 hour meeting with the teaching assistant (and other students who have the same teaching assistant), this would also create a social arena of a kind of mini-class. Students who fail to meet the minimum progress requirements could be obliged to more compulsory learning activities.

Low ambition? As mentioned above, students could achieve an E well before the end of the semester. Many students would be inspired by this and want to progress further to D, C, … However, students could be in a situation with parallel competing interests, and some students might have other courses where they feel much less control and have a genuine fear of failing (say, Math, which many take in parallel with an introductory IT course). Some students may thus prioritize IT less after securing the E, putting more effort in parallel courses instead. However, allowing a student to be content with E and then prioritize another course where the student struggles more to pass, is not necessarily bad. Choosing E in IT, and then passing

Math, too, is presumably better for the student than getting D or C in IT but failing Math, if both are compulsory courses. Hence, giving the student the option to secure an early E can also be framed as a good thing – giving more control back to the student – although it would be sad especially if a student in an IT degree program made such a decision.

4. CONCLUSION

The clear limitation of the paper is of course that it is mere speculation, not reporting any empirical findings. Could PSI be a viable approach for an introductory IT / programming course with a large number of students, or not? Possible advantages is that it gives students more control over their own progress from week to week, and ability to set their own pace and ambition level, which is assumed to be good for student self-efficacy. It gives anyone, from the student who knows advanced programming already, to the one who knows nothing, ability to concentrate on whatever fits their level of competence at the time. At the same time, it is a huge up-front investment and possibly has increased testing costs. Moreover, it is a radical deviation from current dominant teaching practice at Norwegian universities, and would hence be perceived as a high-risk change, especially in a course with many students. An interesting question is therefore if it is possible to implement the shift gradually, rather than as one big operation. The huge question bank might first be developed as an exercise facility, i.e. while keeping a traditional scheme of weekly lectures, compulsory exercises, and final exam – which could then be digital exam with questions in the same genre as the exercises. Then, when the question bank is big enough and questions have also been validated by years of usage, a shift towards PSI could be implemented.

Another reason that PSI appears like a radical deviation is its low emphasis on lectures. Keller (1968) in his early paper "Good-bye, teacher…" envisioned no lectures at all, since lectures were not seen as a good way to present new content (limited learning effect, enforcing the same pace upon all students). Some later PSI implementations have used lectures to some extent, but rather for motivational purposes than to present content. A possibility in our scheme could be to have lectures / seminars / colloquia for various target groups. Our current introductory Python course is run in a traditional way, 1850 students divided into four parallels depending on study programs, and every parallel running a same-pace lecture series throughout the term. With PSI it might instead be possible to divide the student mass according to their current situation, running a seminar for students struggling with unit 3 in room X, another for those struggling with unit 4 in room Y, etc. This would make it easier for the teacher at that seminar, as all students would be at the same level, struggling with understanding the same concepts, and having the same basis (all demonstrated ability to pass the previous unit). Moreover, it should be easier to have more student engagement in such a targeted group, since all students have a clear and immediate motivation for being there, namely gaining the competence for passing that next unit. At the same time, one should not indulge in too many seminars, as many other learning activities (instructional videos, own programming, self-testing etc.) might be equally fit to improve that competence.

REFERENCES

Barnes, D. J. and M. Kölling (2016). Objects First with Java: A Practical Introduction using BlueJ. Harlow, UK, Pearson.
Buskist, W., D. Cush and R. J. DeGrandpre (1991). "The life and times of PSI." Journal of Behavioral Education **1**(2): 215-234.
Caldwell, E. C. (1985). "Dangers of PSI." Teaching of Psychology **12**(1): 9-12.
Crowell, C. R., L. R. Quintanar and K. L. Grant (1981). "PROCTOR: An on-line student evaluation and monitoring system for use with PSI format courses." Behavior Research Methods & Instrumentation **13**(2): 121-127.
Denny, P., A. Luxton-Reilly and B. Simon (2008). Evaluating a new exam question: Parsons problems. Proceedings of the fourth international workshop on computing education research, ACM.

Eyre, H. L. (2007). "Keller's Personalized System of Instruction: Was it a Fleeting Fancy or is there a Revival on the Horizon?" The Behavior Analyst Today **8**(3): 317.

Fleming, R., D. Barth, N. Weber, L. E. Pedrick, S. E. Kienzler and D. M. Reddy (2018). "Effect of U-Pace Instruction on Academic Success, Learning, and Perceptions in Younger and Older Undergraduates." American Journal of Distance Education **32**(1): 3-15.

Foss, K. A., S. K. Foss, S. Paynton and L. Hahn (2014). "Increasing college retention with a personalized system of instruction: a case study." Journal of Case Studies in Education **5**: 1.

Fox, E. J. (2004). The personalized system of instruction: A flexible and effective approach to mastery learning. Evidence-based educational methods, Elsevier**:** 201-221.

Fraser, B. J., H. J. Walberg, W. W. Welch and J. A. Hattie (1987). "Syntheses of educational productivity research." International journal of educational research **11**(2): 147-252.

Gagne, R. M., L. J. Briggs and W. W. Wager (1992). Principles of instructional design. Fort Worth, Harcourt Brace Jovanovich.

Helminen, J., P. Ihantola, V. Karavirta and L. Malmi (2012). How do students solve parsons programming problems?: an analysis of interaction traces. Proceedings of the ninth annual international conference on International computing education research, ACM.

Keller, F. S. (1967). "Engineering personalized instruction in the classroom." Revista Interamericana de Psicologia/Interamerican Journal of Psychology **1**(3).

Keller, F. S. (1968). "Good-bye, teacher…." Journal of applied behavior analysis **1**(1): 79-89.

Kraemer, F. A. (2017). "Team-Based Learning: A Practical Approach for an Engineering Class."

Kulik, J. A., C.-L. C. Kulik and P. A. Cohen (1979). "A meta-analysis of outcome studies of Keller's personalized system of instruction." American Psychologist **34**(4): 307.

Lacroix, M., K. L. McCall III and D. S. Fike (2014). "The Keller personalized system of instruction in a pharmacy calculations course: A randomized trial." Currents in Pharmacy Teaching and Learning **6**(3): 348-352.

Lauvås, P. and K. Raaen (2017). Passion, Cooperation and JavaScript: This is what the industry is looking for in a recently graduated computer programmer. UDIT 2017. Oslo.

Lewis, D. and W. Wolf (1973). "Implementation of self-paced learning (Keller Method) in a first-year course." Journal of Chemical Education **50**(1): 51.

Mayer, R. E. (1989). "Models for Understanding." Review of Educational Research.

Michaelsen, L. K. and M. Sweet (2011). "Team-based learning." New directions for teaching and learning **2011**(128): 41-51.

Parsons, D. and P. Haden (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. Proceedings of the 8th Australasian Conference on Computing Education-Volume 52, Australian Computer Society, Inc.

Pear, J. and W. Kinsner (1988). "Computer aided personalized system of instruction." Machine-mediated learning **2**: 213-237.

Purao, S., M. Sein, H. Nilsen and E. Å. Larsen (2017). "Setting the Pace: Experiments With Keller's PSI." IEEE Transactions on Education **60**(2): 97-104.

Rae, A. and P. Samuels (2011). "Web-based Personalised System of Instruction: An effective approach for diverse cohorts with virtual learning environments?" Computers & education **57**(4): 2423-2431.

Reiser, R. A. (1984). "Reducing student procrastination in a personalized system of instruction course." ECTJ **32**(1): 41-49.

Salleh, N., E. Mendes and J. Grundy (2011). "Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review." IEEE Transactions on Software Engineering **37**(4): 509-525.

Sherman, J. G. (1992). "Reflections on PSI: Good news and bad." Journal of Applied Behavior Analysis **25**(1): 59-64.

Svenningsen, L. (2009). "An examination of the effect of CAPSI as a learning system in developing knowledge and critical thinking in two blended learning courses."

Williams, L., E. Wiebe, K. Yang, M. Ferzli and C. Miller (2002). "In support of pair programming in the introductory computer science course." Computer Science Education **12**(3): 197-212.