A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations

Ian Utting
University of Kent,
UK
I.A.Utting@kent.ac.uk

Dennis Bouvier
Southern Illinois University
Edwardsville, USA
djb@acm.org

Michael Caspersen
Aarhus University,
Denmark
mec@cse.au.dk

Allison Elliott Tew
University of Washington,
Tacoma, USA
aetew@u.washington.edu

Roger Frye
Southern Illinois University
Edwardsville, USA
rfrye@siue.edu

Yifat Ben-David Kolikant
The Hebrew University of Jerusalem,
Israel
yifat.kolikant@mail.huji.ac.il

Mike McCracken
Georgia Institute of Technology,
USA
mike@cc.gatech.edu

James Paterson
Glasgow Caledonian University,
UK
James.Paterson@gcu.ac.uk

Juha Sorva
Aalto University,
Finland
juha.sorva@aalto.fi

Lynda Thomas
Aberystwyth University,
UK
ltt@aber.ac.uk

Tadeusz Wilusz
Cracow University of Economics,
Poland
wiluszt@uek.krakow.pl

ABSTRACT

This paper describes the results of an ITiCSE working group convened in 2013 to review and revisit the influential ITiCSE 2001 McCracken working group that reported [18] on novice programmers' ability to solve a specified programming problem. Like that study, the one described here asked students to implement a simple program. Unlike the original study, students' in this study were given significant scaffolding for their efforts, including a test harness. Their knowledge of programming concepts was also assessed via a standard language-neutral survey.

One of the significant findings of the original working group was that students were less successful at the programming task than their teachers expected, so in this study teachers' expectations were explicitly gathered and matched with students' performance.

This study found a significant correlation between students' performance in the practical task and the survey, and a significant effect on performance in the practical task attributable to the use of the test harness. The study also found a much better correlation between teachers' expectations of their students' performance than in the 2001 working group.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computers and Information Science Education—*Computer Science Education*

ITiCSE-WGR'13 June 29--July 3, 2013, Canterbury, England, UK.

http://dx.doi.org/10.1145/2543882.2543884

General Terms

Measurement, Experimentation.

Keywords

Programming, CS1, assessment, replication.

1. INTRODUCTION

In 2001, an ITiCSE working group led by Mike McCracken (known as the McCracken Working Group and hereafter abbreviated as MWG) met in Canterbury to complete and analyze a study of novice programmers at institutions around the world. The working group produced one of the most highly cited papers in SIGCSE's publication history [18] with two significant outcomes: it demonstrated that CS1 students were less capable programmers than their teachers expected; and it set the scene for a number of subsequent medium- to large-scale multi-national, multi-institutional studies. Despite this, and an explicit call for replication in the original MWG paper, there has been very little effort since directed at replicating or extending the work of the original group.

In 2013, the ITiCSE conference returned to Canterbury and the opportunity was taken to "reconvene" the MWG to address the broad questions of whether "students in 2013 are any more likely to fulfill our expectations than they were in 2001", specifically by:

- critically revisiting the original McCracken study and subsequent work,
- partially replicating their experiment, and
- analyzing and reflecting on the results to determine the extent to which the conclusions drawn by that group are

still valid despite the changes in CS1 teaching and students over the intervening years.

The authors, five of whom were members of the MWG, administered both a practical programming task and a concept assessment test (quiz) as detailed below, and also recorded their expectations of their students' performance. As in the original working group, not everyone managed to collect data, due in part to restrictions on gathering data from their own students, and in part to the timing constraints imposed by operating as an ITiCSE working group (which does not completely form until very close to the end of the European/US academic year).

The task and the test data, along with notes of teachers' expectations, were analyzed during the group's meeting at ITiCSE.

2. METHOD & COHORT DESCRIPTIONS

2.1 Method

To determine the programming ability of a cohort of students from several universities (see Section 2.2 below**)**, the working group devised a two-part assessment. It consisted of a CS1 concept assessment that used The Foundational CS1 Assessment Instrument [24] and a programming skill assessment (the clock problem as described in Section 3). Lastly, we asked the faculty to reflect on their expectations of the performance of their students on the skill and concept assessments and on the actual outcomes of the assessments as compared to their expectations.

The working group decided to use the two-part assessment (concept and skill) to hopefully clarify or better understand the outcomes of the skill assessment. In other words, if a student did well on the skill assessment, did they comparably do well on the concept assessment, and if they did poorly on the skill assessment did they similarly do poorly on the concept assessment? The linkage of skill and concepts is discussed widely in the cognitive science literature (e.g. [19]), in programming cognition literature (e.g. [20]), and examined in recent studies (e.g. [16]).

As became apparent after the fact in the original MWG, teachers entering into studies like this one have a set of expectations regarding the performance of the students on the tasks making up the study. In this study we captured and reviewed these expectations, as described in Section 7.

The concept assessment was a multiple-choice exam and was scored in two ways. A complete score for each student was computed from the 27 questions that cover nine concept areas: Fundamentals, Logical Operators, Selection, Definite Loops, Indefinite Loops, Arrays, Function/Method Parameters, Function/Method Return Values and Recursion. For details on the instrument, its validity, etc., please refer to [23]. A subset score was also computed from the concept areas that the working group determined were used in the skill assessment. That subset of 15 questions was from the concept areas: Fundamentals, Logical Operators, Selection, Function/Method Parameters and Function/Method Return Values. Section 5 describes the concept assessment and it's scoring.

The skill assessment (the clock problem) was scored with a test harness. The harness contained a set of black box tests that validated the functionality of the student's programs as described in Section 3**.** The students wrote their programs in their language of instruction. The languages were Java, Python, and C/C++.

2.2 Cohort

The total cohort for our study consisted of 418 first-year students who have taken at least one introductory programming course at university level. Some students had taken a few other non-programming CS courses, and a few had taken a substantial number. The amount of programming education for the cohort varied between 4 and 10 ECTS with a weighted average of 7 ECTS. (An ECTS credit is a broad measure of student effort, including formal teaching and self-study time. 1 ECTS credit is equivalent to 25-30 hours of student effort [9]).

Members of the working group recruited the students for the cohort. Most of the students were recruited within the institution of the WG member, but some were from other institutions. Overall, the cohort represents students from 12 institutions in 10 countries. 18% of the cohort is from the USA, and 82% is from Europe. Approximately 50% of the total cohort is from a single European university.

For organization of data collection and analysis, we divided the cohort into eight groups. The groups vary in many ways, e.g.:

- amount and type of programming education prior to data collection
- amount of additional non-programming CS education
- program of study (CS major, CS minor, Engineering, CS/programming as an elective, etc.)
- type(s) of programming language(s) used
- language of instruction (native/foreign)

Table 1 provides an overview of the eight groups in the cohort and which parts of the assessment they took part in, although not all students who attempted both parts completed both parts.

Table 1: Overview of Cohort. "Credits" reflect the volume of study, measured in ECTS credits

Groups	Course sections	Study program	Language and style	Programming Credits	Other CS Credits	N	Concept test	Clock test
R1	1	Eng.	Python(proc.)	5	0	151	x	x
R2	1	Eng.+CS	Python (OO)	10	0-10	58	x	x
P	1	CS	Java (OO)	10	25	26	x	x
T	8	Mostly CS	Java (OO) and C (proc.)	10	20	57	x	x
V	1	CS	C# (proc.)	~10	?	17	x	
Q	2	CS	C/C++ (proc.)	4	21	49	x	x
S	1	Eng.	C++ (proc.)	6	0	40	x	x
U	1	CS	Java (OO)	8	0	20		x

3. THE CLOCK TASK

As a test of programming ability, students were asked to undertake a simple programming task. A reference implementation was written in Java, with the instructions to students embedded in comments in the code.

3.1 The Problem

Students were asked to complete the implementation of a class (called Time) representing a 24-hour clock. The behavior of the clock with respect to wrap-around of the hours, minutes and seconds values was described with examples. The clock has four operations which students were asked to implement: a tick operation which advances the stored time by one second, a comparison operation which determines the order of two times, and add and subtract methods which calculate the sum or difference of two time values.

The problem is designed to focus on students' ability with the concepts of selection, arithmetic and Boolean expressions, although it also touches on their understanding of method parameters and return values. Unlike the original MWG, it is also designed only to require students to implement a part of a complete program, with a strong bias towards ADT implementation, rather than the original algorithm-focused input-parse-output-loop style.

Students undertook the task in "closed lab" settings of around 90 minutes duration (group S had 75 minutes, and group U 110). For most groups, the task was completed as part of a course and most of the students on that course undertook it. In three cases (R1, R2 and P) the students were volunteers comprising 10-30% of their respective cohorts. Analysis of their performance in the courses from which they were recruited suggests that the volunteers were representative of their cohorts. Most of the participants were mildly incentivized to participate, either by entry to a small-prize raffle (T,S and U), a coursework grade bonus (Q) or a small prize and a grade bonus (R1 and R2). Group P students received no incentive.

3.2 Reference Implementation

The reference version of the task was written in Java, and translated into other (implementation) languages by individual investigators. The instructions for undertaking the task were included (as comments) in a skeleton implementation (Time.java) provided as a starting point for students (see Appendix A). This skeleton included the class boiler-plate code down to the level of skeletons of the required methods as well as the descriptive comments. As well as the skeleton code, the reference implementation included an example solution and a test harness. The intention was that the test harness and skeleton implementation of the Time class should be provided to students as a starting point and a check for their work. In some institutions however, the test harness was not provided to students, although it was subsequently used to assess the accuracy of their implementations.

The skeleton of the Time class included full implementations of the entire class, with the exception of the bodies of the methods students were required to complete. In the case of the comparator method the skeleton body included a return statement to ensure that the skeleton compiled. As well as these methods, the skeleton also included constructors and a toString() method (to produce a printable representation of the time value) to support testing of the implementation.

The reference implementation also included a test harness containing 8-10 black-box tests for each of the four methods to be completed in the Time class. These tests covered both simple cases; all of the rollover cases for the tick() and add() methods, and the "borrow" cases for subtract(). In addition, the Java and OO Python implementations included tests for common implementation problems (e.g. equality/identity confusions).

The test harness was organized so that all the tests for a particular method were performed, independent of any failures, but tests for subsequent methods were only executed if all prior tests had passed. This was intended to avoid presenting students with a long list of failure messages before they had started their work, but had the effect of "ordering" students' approach to the tasks. Students using the test harness were discouraged from working on methods before all the tests on "previous" methods passed. The order of method tests in the reference test harness was: tick(), compareTo(), add(), and subtract().

3.3 Translations

The reference implementation was translated into C/C++, C# (which was eventually unused) and two variants in Python. In addition, the comments in both Python versions were also translated into the local (natural) language for use in one of the institutions. Other institutions where the students' first language was not English nevertheless used the English versions of the instruction/comment. All versions are available on request from the first author.

4. THE TASK: ANALYSIS & RESULTS

4.1 Analysis

The participants' Clock Task submissions were evaluated using black-box tests. Using the same four sets of tests – one for each method that needed to be implemented – provided to most of the participants as part of the programming task. However, in evaluating submissions, all tests were run, even if an earlier test had failed. A method in one submission was judged completely correct if it passed all of the tests for that method; passing only some of the tests for a method was a failure. Combining the results for each method in a submission determined the overall mark for that submission, which is an integer between 0 and 4 – a count of how many of the methods in a submission passed all the tests.

4.1.1 Results

Table 2: Results on the Clock Task

Cohort	Test Harness?	Number of students	Average # of methods working	Success by method (%)			
				tick()	compareTo()	add()	subtract()
R1	Yes	149	3.04	82	81	72	68
R2	Yes	57	3.86	98	98	96	93
P	Yes	26	3.27	92	92	73	73
T	Yes	38	3.21	84	89	76	71
Q	No	15	0.80	33	13	27	7
S	No	40	0.93	33	29	17	14
U	No	20	0.65	30	15	10	10
combined	Yes	270	3.26	87	87	78	74
combined	No	75	0.83	32	22	17	12

combined	All	345	2.72	75	73	65	61

Table 2 shows, for each cohort, the average number of methods successfully completed by the participants as well as the percentages of participants who successfully completed each individual method. On average, the participants completed 2.72 methods out of 4. This overall average leans towards the larger cohorts however, and as is obvious from the table, there were substantial differences between the cohorts, with a group of cohorts scoring very high and another group very low. As Table 2 also illustrates, a significant factor in this two-way split appears to be whether the cohorts had been provided with a test harness or not. In all cases, it was reported that students had previously been exposed to ideas of testing software, but had not been asked to take a systematic approach to it in their work. Below, we will discuss the results of the two groups separately.

4.1.2 Cohorts with a Test harness

In four cohorts (R1, R2, P, T), the students completed an average of 3.26 methods out of four, with the majority of students completing all four. In all of these cohorts, the students were provided with a test harness as described in Section 3.

The test harness strongly encouraged the students to attempt each method in order and not skip ahead before they had a working solution to the previous method. It is unsurprising; therefore, the first method (tick()) was correctly implemented more than the other methods, with the number of successful submissions decreasing at each successive method.

Table 2 suggests there were two points in the four-method sequence that caused some of the students to get stuck and not make further progress. Some fell at the first hurdle: about 13 % of the with-harness students could not produce a working implementation of the tick() method. Nearly all of those who succeeded with tick also did well on the next method, compareTo(); in one cohort (T), the result was better for the second method than the first. The second spot of difficulty arrived with the third method, add(); about 9 % of the students failed at this point, but those who succeeded went on to produce a fully working solution to the last method, subtract(). The correlations between methods shown in Table 3 bears out this interpretation: success in implementing tick() and compareTo() correlate relatively strongly with each other, as do add() and subtract() with respect to each other. These results suggest that the students found the first two methods to be easier than the other two.

Table 3: Correlation between Performance on Sub-tasks in Clock Task, calculated for each student

Sub-task	tick()	compareTo()	add()	subtract()
tick()	1	.500	.354	.352
compareTo()		1	.262	.391
add()			1	.591
subtract()				1

4.1.3 Cohorts without a Test harness

Due to ambiguity in the methodology as explained to working group members, participants at some institutions were not provided the 'test harness'. Some of these participants were only given the Time class; others were given the Time code with a main method, but no test cases.

In contrast to the cohorts discussed above, submissions from participant cohorts not provided with the test harness (i.e., cohorts Q, S, and U) have an average of 0.83 correct methods.

A few participants left traces of creating their own test harness in their submissions, others may have created testing facilities but not submitted them. Evidence suggests that less than 5% of the students did any systematic testing. However, even in the absence of a test harness, many student code submissions pass many of the unit tests for one, or more, methods.

Not having the test harness requires students to identify and correctly implement all the corner cases, as well as avoiding inserting any unrecognized bugs of their own.

The possible implications of not having the test harness:

- It requires participants to understand the use of the Clock class from its documentation alone, rather than from the examples provided by the harness.
- It requires participants either to create their own test cases, or not test their work at all.
- It requires participants using OO languages to realize the Time class will be used by an object of another class, which might be a novel approach for them.
- The harness imposes an order of work – non-harness students may lack scaffolding without the ordering imposed by the harness.
- A failing test in the harness may discourage students from moving on to subsequent sub-tasks.

Mistakes (made less likely with the harness) seen in non-harness participants (cohort U):

- including a main() method (2/20)
- creating a loop in the tick() method (3/20)

Observing that several students in the no-harness group have partial solutions, an alternative analysis of these submissions was devised. The same unit tests were run for the no-harness submission. However, instead of recording a binary success / fail for each method, the numbers of tests passed for each method were tallied. Table 4 summarizes the results.

45 of 75 (60%) of the submissions were judged partially correct code whereas only 3 of 75 (4%) of the submissions were judged completely correct.

Table 4: Detailed success rates for non-harness students (n=75)

Sub-task	**Partial success (%)**	**Complete success(%)**	**Total (%)**
tick()	19	33	52
compareTo( )	37	23	60
add()	21	19	40
subtract()	21	12	33

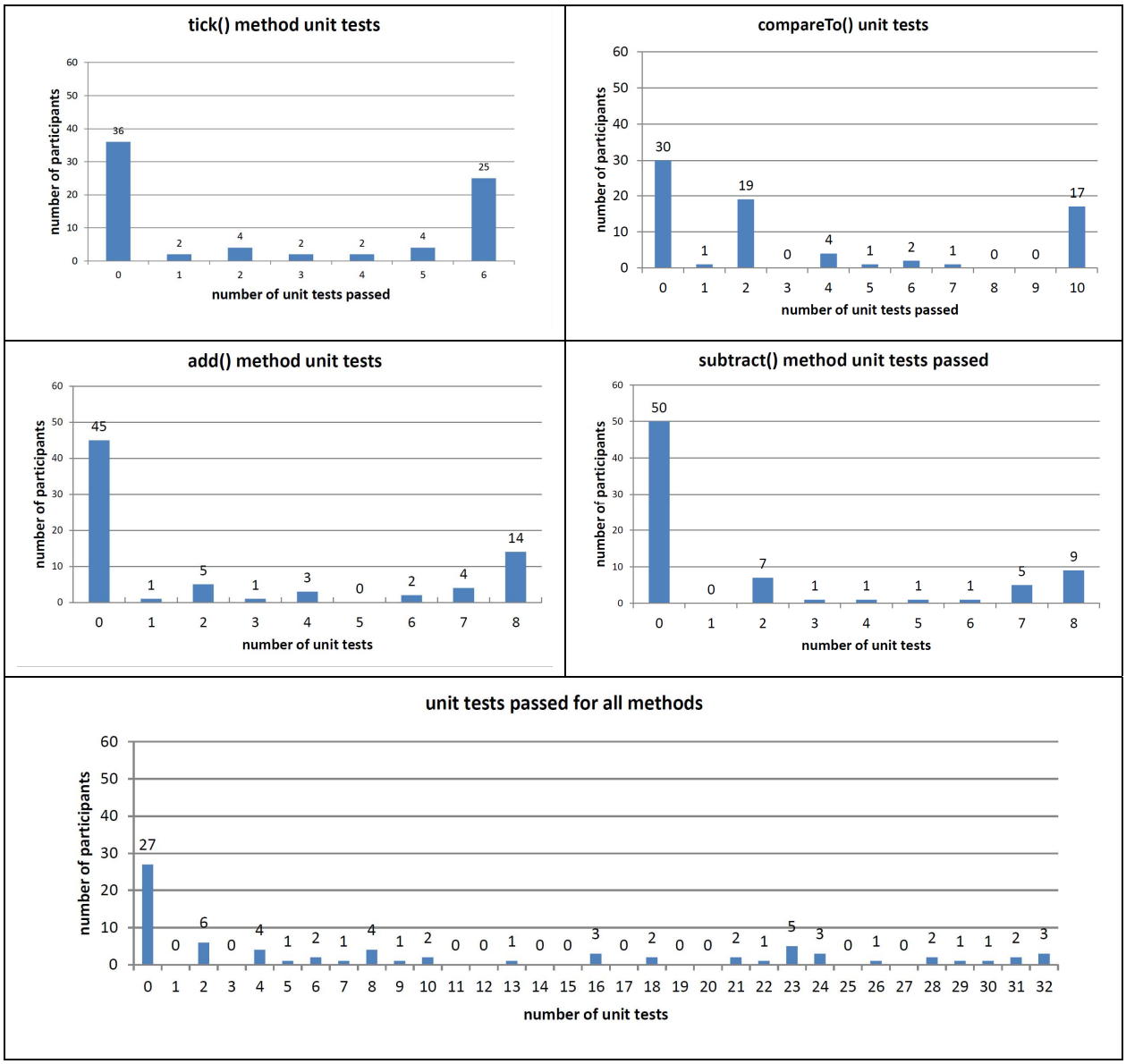

Figure 1: Partial success for the non-harness groups

Figure 1 shows histograms of the number of test cases passed by students not provided with a test harness for a) the tick() method, b) the compareTo() method, c) the add() method, d) the subtract() method, and e) all methods combined. As can be seen from this figure, students in these cohorts had a tendency to either pass none of the tests for a particular method, or pass all of them. This leads to an apparent bi-modality in the outcomes at the method level, which is not apparent at the aggregate level. This probably represents a relaxation of the ordering imposed by the test harness, with students here successfully completing some, but not all, method implementations. The "spike" in successful completion of the compareTo() method (with 2 successful unit tests passed) is an artifact; the skeleton provided for this method coincidentally passes two of the tests.

It should be noted that:

- 3 of 75 submissions passed all N tests
- 42 of 75 submissions passed between 1 and N-1 tests
- 22 of 75 submissions passed at least one unit test for each method

In these cohorts, too, a trend can be observed in that the students were more successful with the first methods than the later ones, although there is more variation in this respect in the no-harness group than in the with-harness one. This greater variation is likely to be a reflection of the no-harness students being less constrained in their choice of which methods to tackle and when. It may be that the order of appearance of the methods in the provided skeleton, which was the same as the order of the method tests in the test harness, suggested an implicit order in which students attempted implementation.

5. THE ASSESSMENT INSTRUMENT

5.1.1 Assessment of Conceptual Understanding

The Foundational CS1 (FCS1) Assessment Instrument was used to measure students' conceptual understanding of programming concepts [24]. The FCS1 is a validated exam of topics commonly found in a first computer science course and is written in pseudo-code so that it can be used in courses that use a variety of programming languages and pedagogies.

The exam uses a multiple-choice question format to investigate topics in three different dimensions: definition, tracing, and code-completion. The definition questions explore a student's understanding of a concept, while the tracing questions ask students to predict the outcome of the execution of a piece of code. Code-completion is the code-writing task, where students are asked to fill in missing portions of code to complete a function to produce a certain result.

The validity of the assessment instrument has previously been demonstrated using a three-pronged mixed methods approach integrating both quantitative and qualitative techniques. Think aloud interviews provided evidence that students were reading and reasoning with the pseudo-code to answer questions in the manner intended. Statistical analysis techniques demonstrated both the quality of the questions themselves as well as a correlation with external faculty definitions and measures of CS1 knowledge [24].

5.1.2 Data Collection & Analysis

The FCS1 was administered via a web-based survey tool at six different universities. The exam was given under testing conditions – a closed laboratory setting with proctors to supervise the testing environment. Students were given one hour to complete the assessment, and the majority (96.1%) finished within the time limit, or at least did not appear to have run out of time[1]. A two-page overview of the pseudo-code syntax was provided to each student before the exam began and was available for reference throughout the assessment.

5.1.3 Results

We received a total of 231 valid responses to the FCS1 assessment. Before data analysis could begin, outliers from the data set that would bias or skew the results were removed. Exclusionary criteria include: empty submission, entered the same answer to 10 or more questions in a row, and spending less than 15 minutes on the entire exam (an average of 33 seconds per question.) A second researcher verified the rules for exclusion and independently reviewed all of the exams that were removed from the data set to confirm that they met one or more of the exclusionary criteria. After scrubbing, the final data set consisted of 217 responses.

The FCS1 was then scored, awarding a 1 for a correct answer and a 0 for an incorrect answer. (Any question left blank was not scored.) The maximum score was a 25, and the minimum score was a 2 out of a total of 27 questions. Student participants answered an average of 11.35 (42.02%, σ = 4.711) questions correctly. Questions about math operators and if statements were among the most commonly answered correctly. The programming constructs related to function parameters, function return values, and definite loops were the most difficult questions. The distribution of performance on the concept assessment by cohort is shown in Table 5. There was a statistically significant difference between groups as determined by one-way ANOVA ($F(6,210)=23.119$, $p = 0.000$). A Tukey post-hoc test revealed that cohorts R2 and P scored significantly higher than all of the other cohorts (16.81 and 16.36 respectively). Further, the Tukey post-hoc test identified a subset of cohorts (R1, T, and Q) that performed better than the remaining two cohorts. There was no statistically significant difference between the remaining two cohorts ($p = 1.000$).

Table 5: Overall Student Scores on the FCS1 Assessment Instrument by Cohort

Cohort	N	Average	%	σ	Median
R1	15	11.27	41.73	3.97	11
R2	16	16.81	62.27	4.56	17
P	25	16.36	60.59	4.23	15
T	57	12.02	44.51	4.08	12
V	17	7.53	27.89	3.47	7
Q	49	10.31	38.17	3.38	10
S	38	7.69	28.49	2.68	8

A subsequent analysis examined the performance of students on the subset of topics on the FCS1 assessment that were identified as learning objectives in the clock task skills assessment: fundamentals, logical operators, selection statement, function parameters and function return values. The maximum score was a 14, and the minimum score was a 0 out of a total of 15 questions. Student participants answered an average of 5.96 (39.76%, σ = 2.657) questions correctly.

Table 6: Student Scores on the FCS1 Assessment Instrument on Task Topics by Cohort

Cohort	N	Average	%	σ	Median
R1	15	5.47	36.44	2.45	5
R2	16	8.81	58.75	2.81	8.5
P	25	8.64	57.60	2.66	8
T	57	6.33	42.22	2.42	6
V	17	4.76	31.76	1.64	5
Q	49	5.29	35.24	1.86	6
S	38	4.10	27.35	1.79	4

Questions about math operators and logical operators were among the most commonly answered correctly. The programming constructs related to function parameters and function return values remained the most difficult questions. The distribution of performance on the concept assessment by cohort is shown in Table 5. There was a statistically significant difference between groups as determined by one-way ANOVA ($F(6,211)=17.168$, $p = 0.000$). A Tukey post-hoc test revealed that cohorts R2 and P scored significantly higher than all of the other cohorts (8.81 and 8.64 respectively). Further the post-hoc analysis identified that cohort T participants performed significantly better (6.33 ± 2.42

[1] A participant was determined to have run out of time if they worked on the assessment for the full hour and left a significant percentage (>35%) of the questions at the end blank.

points, p = 0.000) than the S cohort. There were no statistically significant differences between the remaining cohorts.

6. CORRELATIONS BETWEEN THE ASSESSMENT AND THE CLOCK TASK

6.1 Overall Task Score and Concept Assessment

A Pearson product-moment correlation coefficient was computed to assess the relationship between the scores on the skills and conceptual assessment instruments as enacted by the clock task and the FCS1 assessment instrument respectively. There was a positive correlation between the two variables, r = 0.653, n =140, p = 000. Overall, there was a strong, positive correlation between the overall score on the clock task (i.e. the number of tests a student passed) and their score on the FCS1 assessment (see Figure 2). Further, there also exists a strong positive correlation between the clock task score and the score on the subset of the topics isolated by the task (r = .605, n = 141, p = .000). See Table 7 for more details.

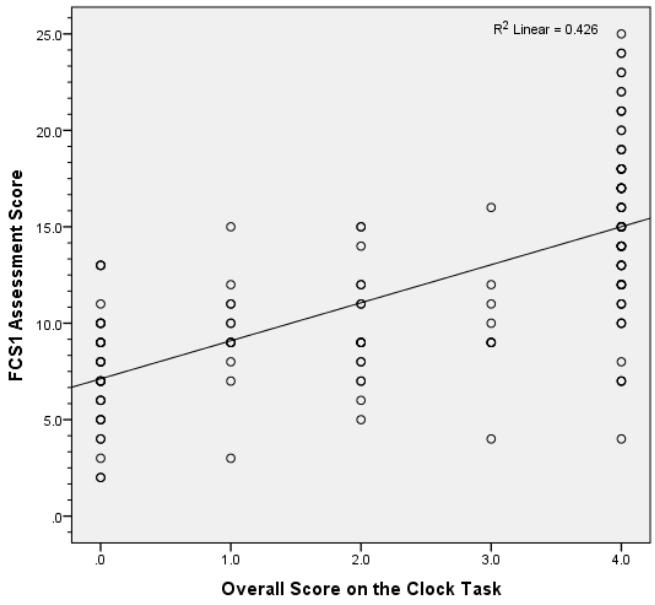

Figure 2: Graph of students' overall score on the Clock task vs score on the FCS1 Assessment for the overall population

Subsequently, in order to investigate the extent to which the test harness mediated task performance, we conducted another correlation study with the total population split into two subgroups by whether or not they conducted the clock task assessment with the test harness A Pearson product-moment correlation coefficient was computed to assess the relationship between the scores on the skills and conceptual assessment instruments as enacted by the clock task and the FCS1 assessment instrument respectively by subgroup.

Table 7: Pearson's Correlation between Clock Task Score and Concept Assessment Score

Cohort	N	Overall FCS1 Score		Task Topics FCS1 Score	
		r	*p*	*r*	*p*
Total	140	.653	.000	.605	.000
With Test Harness	89	.473	.000	.403	.000
Without Test Harness	51	.287	.041	.392	.004

There was a positive correlation between the two variables clock task score and assessment score for both subgroups. However, the decrease in correlation (r = .473 and r = .403) suggests that the test harness is indeed scaffolding students' performance, perhaps beyond their ability to fully understand the conceptual material exercised in the skills task. The weaker correlations in the "without test harness" subgroup are likely caused by the very strong floor effect in the task performance (see Figure 1). A general view of students' relative performance, separated by the availability of the test harness is given in Figure 3 and Figure 4.

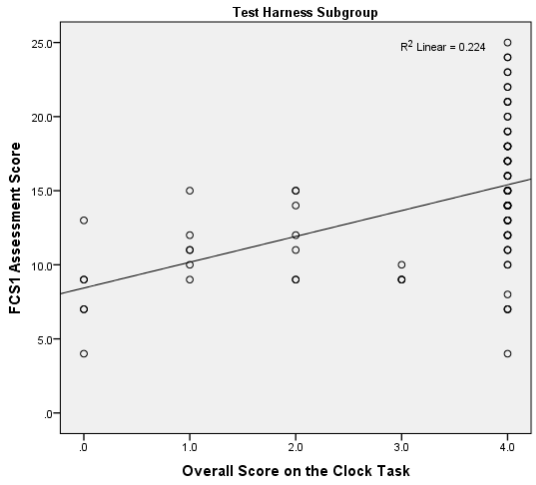

Figure 3: Graph of students' overall score on the Clock task vs score on the FCS1 Assessment for students with a test harness.

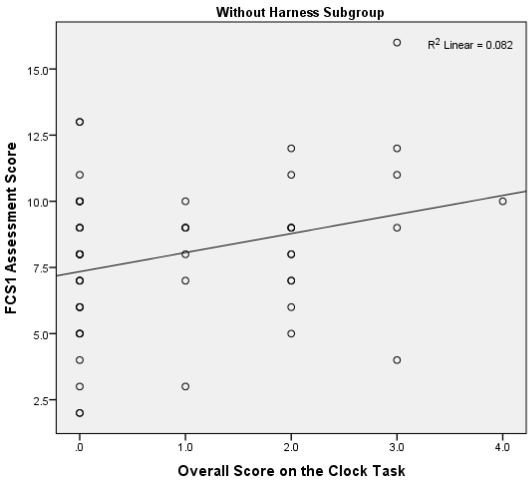

Figure 4: Graph of students' overall score on the Clock task vs score on the FCS1 Assessment for students without a test harness

6.2 Sub-task Test Score and Assessment

As described in Section 4.1.3, in order to give participants without the benefit of the task harness an opportunity to demonstrate their level of programming skill, the clock task was rescored awarding one point for each unit test passed rather than an overall pass/fail score if they had successfully completed all of the unit tests. The results of this more detailed scoring were used to assess the relationship between the scores on the skills and conceptual assessment instruments for those participants who were not given the testing harness.

A Pearson product-moment correlation coefficient was computed. There was a positive correlation between the two variables, r = 0.292, n = 48, p = .044. Overall, the results are similar to those found with the simplified scoring scheme. There was a positive correlation between the overall score on the clock task and their score on the FCS1 assessment**.** Further, there also exists a somewhat stronger positive correlation between the clock task score and the score on the subset of the topics isolated by the task (r = .396, n = 49, p = .005) and no significant correlation between the clock task score and the score on the subset of the topics that were deemed outside of the scope of the task.

Further investigation is needed to fully understand the extent the differences in the ways these two subgroups performed in the clock task. However, the fact that the correlation of the task and assessment scores on both task and non-task isolated topics was similar (.403 and .454 respectively) suggests that while the test harness clearly scaffolded performance on the clock task it did not mask students' latent understanding of the conceptual content highlighted in the task.

7. TEACHERS' EXPECTATIONS

As mentioned above the group members who brought data cohorts were asked to fill in a short survey and describe their predictions regarding the performance of their students in the FCS1 assessment. They were also asked to posteriori reflect on their expectations and the actual performances of their students in the Clock programming task as well as the FCS1. Although all teachers were very familiar with the original MWG work, only one of them gathered data in both the original study and this one.

7.1 Expectations regarding the FCS1

The teachers were asked to predict the overall score of their students (0%-100%) in the FCS1, the topics and subtask that were easiest and most difficult to their students (they had to choose from the following list: Fundamentals, logical operators, Selection, Definite Loops, Indefinite Loops, Arrays, Function Parameters, Function Return Value, Recursion), and the percentage of students who might have run out of time. Table 8 presents the teachers' response regarding the foundational CS1 assessment instrument (FCS1)

Table 8: Teachers' estimations on their students' success in comparison to students' performance for the FCS1

	Anticipated Score	Score	Easiest Topic	Most Difficult Topic
R1	44%	Realist (2%)	Fundamentals & Selection	Recursion
R2	63%	Realist (1%)	Fundamentals & Selection	Recursion
P	42%	Pessimist (19%)	Fundamentals	Recursion
T	56%	Optimist (12%)	Selection	Recursion & Arrays
Q	52%	Optimist (14%)	Selection	Recursion
S	40%	Optimist (12%)	Fundamentals	Recursion

All the teachers unanimously thought that Recursion is the most difficult topic. Fundamentals (i.e. variables, assignments, and so forth) and selections were considered to be the easiest topics.

Teachers' estimations were in the range of 40-63% (in literature the average was 42% [24]). About half of the teachers' believed that the students' conceptual knowledge was better than it actually was (see Table 5 and Table 6).

In their reflections, teachers mentioned several concerns regarding factors that might have influenced students' performance in the assessment. Two teachers were concerned that students did not have sufficient time. Another teacher was concerned about the students' limited knowledge in English, the language of the test and the task. Another concern mentioned by one teacher about a possible cultural bias was that the concept of a 24-hour clock would be difficult to his students, who are accustomed to a 12-hour clock.

Two teachers had concerns about their students' conceptual knowledge. One was concerned that their students had not been exposed to some topics, such as recursion. Another teacher stressed that, in his institute, they "prioritize practical programming skills and techniques over deep conceptual understanding".

The teachers mentioned that they were familiar with the literature relevant to the FCS1 assessment. This has "colored" or "biased" their expectations.

7.2 Expectations from the Clock programming task

The teachers were asked to rank the four sub-tasks from the easiest (score of 1) to the most difficult (score of 4). Table 9 presents their ranking. The majority agreed that the subtract() sub-

task would be the most difficult and that the add() sub-task would be the second most difficult.

Their explanations emphasized the relations between these two sub-tasks. The most common argument was that add() is "algorithmically more complicated" than tick() and compareTo(), and subtract() is like "the add() function in reverse but will cause students more difficulty[.]".

Table 9: Teachers ranking of the sub-tasks

	tick	compareTo	add	subtract
R1+R2	1	2	3	4
P	1	2	3	4
T	2	1	3	4
Q	4	1	3	2
S	1	3	2	4
U	2	1	3	4
Average	1.83	1.67	2.83	3.67
σ	1.17	0.82	0.41	0.82

The tick() and the compareTo() sub-tasks were considered by the majority to be easier than the add() and subtract() sub-tasks because tick() requires "a simple manipulation" and compareTo() requires a "relatively simple code". They varied, however, in their choice between the two. These estimations were correct, as can be seen in from Table 3.

Teachers were also asked to estimate their a priori expectation of complete success in the Clock task: what proportion of their students they expected to be able to completely implement the task (to the point of passing all tests), with the results given in Table 10.

Table 10: Teachers' estimations on their students' success in comparison to students' performance for the Clock task

	Overall	estimation-overall
T	66%	50%
S	3%	4%
R1	68%	80%
R2	93%	99%
U	10%	10%
Q	0%	5%
P	73%	33%

8. COMPARISON WITH THE ORIGINAL MCCRACKEN WORKING GROUP AND OTHER, MORE RECENT, STUDIES

In this section we compare the original McCracken working group study [18], the Sweden Group study (SG) [17] and this study[2].

The MWG study conducted a multi-national, multi–institutional study in which the students were given one of three related calculator exercises which were deemed to cover all parts of the learning objectives framework. Two measures were used to evaluate the students' attempts: a General Evaluation (GE) which included execution, verification, validation and style components, and a Degree of Closeness evaluation (DOC) in which the code was examined qualitatively. Overall the MWG "found that the students' level of skill was not commensurate with their instructors' expectations". As measures of this we note that the average GE score (which was mainly objective) was 22.9 out of 110.

The SG took the MWG as its starting point and gave the infix, precedence-free calculator problem to 40 students at one institution. The study addressed three research questions. The first of these, and the goal most relevant to this paper, was how well can the students at *one* institution solve a calculator problem if they do not have to deal with various confounding issues presented in the MWG study (unfamiliar environments and conditions, the complex explanation of calculators, the need for a stack for the postfix calculator, the complexities of Java I/O, and no access to an online Java API). The SG results were much more encouraging than in MWG. The GE score average was 68.2 out of 110. The authors conclude that "generally the students were able to do at least part of the problem". They offer several possible explanations for these different results, which relate to the specific issues mentioned above, and which they categorize as environment, cognitive load and troublesome knowledge.

Another research question from the SG study is "can a modified version of the instrument used by the MWG provide a useful assessment?" They refer to MWG as having the goal of evaluating its instrument, as well as the students. We think this is a slight misunderstanding, since MWG asked participants to choose students who "should" be able to solve the calculator problem. The MWG question was rather: "are instructors' expectations of their students realistic?"

In the current paper we are reporting on a study with similar goals to the original MWG but with different assessment instruments, including a (programming) language neutral test of students' conceptual understandings. The cohort for this study is larger than both MWG and SG.

Both MWG and SG used calculator problems. This study used the clock problem, which was considered to be more in line with object-oriented environments and less algorithmically complex. A more important difference between MWG, SG and this study is that MWG involved a problem that tested all the parts of the learning objectives framework which they identified. In particular, the first two parts (abstracting the problem and generating sub-problems), which have been noted by SG and

[2] A note on dates: the original MWG met in 2001, SG produced their paper for ICER 2013, but this group was able to see a preprint just before we met.

other researchers [16] as too high an expectation. SG gave the participants a skeleton calculator and some I/O code as a starting point. Our study also gave a quite explicit skeleton Time class and the students were told to fill in the method bodies. A testing framework was also made available to some of the students that may have given them some scaffolding and a stopping criterion. Thus, in both the SG study and the current study the design was essentially given to the students.

In summary, MWG raised the whole issue of student performance and instructor expectation in a multi-national context. The SG study was able to remove many of the issues that confounded the original MWG study. They showed that expecting students to be able to code a no-precedence infix calculator with considerable amounts of scaffolding code with a smaller cohort at a single university produced better results than MWG. The current study shows that instructors' expectations appear to be more accurate than in the era of MWG, that student performance is better when students need not design a whole application and are able to easily verify their results, which coincides more closely with the SG study results than the MWG study.

9. THE WORLD IS DIFFERENT

In the 12 years between the original MWG and this study the world as experienced by this digital generation of CS students has changed. Many changes have taken place in the way we teach and assess our students, and students themselves have changed in terms of the prior knowledge they bring with them and of the way they discover information and solve problems.

Changes in teaching and assessment

One of the issues identified as central to the effort of the MWG was the development of CC2001[8]. The current study takes place within the context of the development of CS2013[14] which reviews and enhances CC2001 and the interim CS2008[25]. The draft documentation for CS2013 identifies a number of interesting features of the evolution of introductory CS courses from CC2001 to CS2013:

- Increase in the prevalence of "CS0" courses and multiple pathways into and through the introductory course sequence;
- Growing diversity in platforms used, e.g. web development, mobile device programming;
- Broadening of the list of languages used, and trend towards managed and dynamic and visual languages, with no particular paradigm or language becoming favored;
- Increasing adoption of software engineering innovations, e.g. test-driven development[2], version control, use of IDEs.

These features were reflected to a limited extent within this study. There was some diversity in the pathways through the course sequence among our cohorts, which presented difficulties in comparing cohorts. All students participated in the task using desktop or laptop computers, which matched the environments they were accustomed to using. The only addition to the languages used in the MWG was Python, while other cohorts used Java and C/C++ as in the earlier study. The MWG acknowledged the possibilities of Test-driven Development/Design (TDD) approaches to allow students to check work at an earlier stage, and the design of the task in this study made use of TDD techniques for scaffolding the students' activity in some cohorts. However, by no means all the students in this study had experience in their courses of developing software using a TDD approach.

In addition to the features identified in CS2013, developments arising from CS education research have had an impact on teaching and learning, for example:

- Transition from written exam with pen and paper to practical exam with computer and the development tools and resources which students practice with in labs;
- Transition from procedural to object-oriented programming;
- Recognition of the roles of variables [21];
- Transition toward a systematic and structured focus on constructive alignment between intended learning outcome, course activities, and assessment[6], including instances of assessments specifically designed to address issues raised by the findings of the MWG[4].

Modern object-oriented programming languages come with a large class library and a well-documented API to ease access to it, and many educators take advantage of the opportunity to produce partially finished programs and/or provide classes as black boxes for the students to use when solving the problem

In earlier days it was less common to use libraries and typically students built everything from scratch when they were programming. With object-oriented programming entering the stage, it has become much more customary for students to contribute to already existing code either by using standard "slave" classes that offer various "low-level" functionality (typically the Model in an MVC structure) or by using frameworks that provide an overall structure where the students contribute by concretizing hot spots in the framework (by implementing virtual methods/subclasses)[7].

The building blocks that are given to students may be provided as black-box or white-box components [13]. The former refers to components that the students are supposed to use by only referring to the specification of the components whereas the latter are components that the students must open, read/study, and modify. In the former case, the consequence is that students read more APIs (specification level). The latter case has as a consequence that the students read more code (implementation level). Overall, students tend to spend more time studying existing code now than they did when the MWG study was conducted.

Of course, while the above observations reflect identifiable trends, such developments are not universally accepted and practiced, as evidenced, for example by a recent discussion on the SIGCSE mailing list regarding written coding questions in examinations, where educators expressed significant support for requiring students to hand-write code without access to documentation or syntax-checking[22].

Changes in the students

Today's students have been described as belonging to the Net generation, or as Digital Natives[15], who are active experiential learners, dependent on technology for accessing information and interacting with others. These students may not readily engage with the instructional resources, such as textbooks, available to previous generations. The existence of this generation has been disputed, however, and it is not clear that a particular learning style or preferences can be attributed to a whole generation [5]. Nevertheless, while there is little evidence that the level of CS instruction experienced by students before coming to university

has increased significantly, it can be argued [12] that the increasing use of computers, software and online resources among young people leads students to have developed theories of how computing works by the time they start their CS courses.

This type of digital literacy is from a user or user-programming rather than a professional viewpoint [3], but it carries over into some specific expectations when students are programming. CS students now expect "always on" access to the Internet as they code and this is likely to influence the strategies which they adopt in attempting to solve programming problems. Instructors and textbooks (e.g. [1]) often encourage use of online programming language API documentation while developing programs, as good software engineering practice. There are many more online spaces where students can seek answers to specific questions from instructors or peers within an educational context, for example Piazza [11] or from members of the wider developer community, such as StackOverflow (http://stackoverflow.com). The first instinct of many programmers, students or otherwise, is probably to search for an answer on Google, which in turn will often find answers to similar questions which have been asked previously. More general social media, such as Facebook and Twitter, are also widely used by students and may be used to seek support.

The influence of these strategies on the activity within this study is difficult to determine as this behavior was not explicitly recorded or observed. The task, designed to translate easily to different programming languages, requires little or no use of API classes or functions, so online documentation would have been of little use here. The task was administered in a time-limited context, albeit with no restriction on access to the Internet. It is unlikely, though not impossible, that a student would be able to pose a question and receive an answer online within that timescale. This consideration influenced the choice of the problem for the task, as we searched to assure ourselves that this was not a findable problem with "canned" solutions readily available.

10. DISCUSSION

Overall, students seem to have performed better on the programming task used in this working group than in the one used in the original MWG. In fact, the low-scoring "no harness" groups in this study performed as well (on the "passed all tests" measure) as the average "general evaluation" score of 21% across all cohorts in the MWG.

That having been said, there are clearly two distinct populations within the current study's overall cohort: one with an average completion rate of >3.0 methods, and one with an average < 1.0 (Table 2). There are a number of potentially significant factors involved in this difference:

Some of the cohorts in the high-average group had more prior programming material in their University education than others. That is: the size of the "CS1" component at the end of which they participated in this study varied from 5-10 ECTS credits (Table 1). Discounting any pre-university programming experience, this means that some students had twice as much exposure to (and practice in) programming before attempting the task.

Some groups undertaking the Clock task were provided with the test harness. This clearly had an effect on their performance in the Clock task, as shown by the correlations with their performance in FCS1 (Table 7). We believe that this explained by a scaffolding effect:

- The test harness guides students in what they need to do:
    - It serves as a definition of correctness for the students: what is a correct solution like?
    - It disambiguates requirements that may have otherwise been unclear: does tick() mean a single tick or making the clock tick continuously?
    - It reminds the student of corner cases that they may otherwise overlook.
- Assuming the student uses the harness, they receive continuous, instant feedback about their program.

On a related note, students sometimes choose not to write tests early, even when taught using practices such as TDD (e.g. [10]). It is apparent from inspection that few, if any, in the no-harness group wrote a set of tests and then implemented the required methods; consequently, they would not have had access to feedback as they worked incrementally on the four methods.

Being given a test harness also meant less work and less mental load for the students:

- The harness takes care of I/O.
- The harness provides a main method and removes the need for the students to design any of the overall structure of their program, which represented two parts of the MWG learning objectives framework.
- Having the harness simply means that there is less implementation work to be done: the student does not need to write tests, is less likely to run out of time, and is less likely to suffer from time pressure.

Assuming that we are correct in stating that the four methods of the Time class were in more or less increasing order of difficulty, then the harness also suggested or even enforced an effective path from tick() to subtract() so that implementing each preceding method makes the next one a smaller step in difficulty. Sequencing learning activities on a topic in order of increasing complexity helps keep the students' cognitive load in check [26].

Writing the tests is likely to have been difficult for some students. Some aspects of test-writing may even have been conceptually more difficult for them than aspects of the task proper. For instance, using the Time class from the test code requires an understanding of object-instantiation that is not required to implement any of the methods in the Time class itself.

Finally it is worth commenting on our results in looking at teachers' expectations of their students' performance. In the MWG the teachers were all negatively surprised: "the first and most significant result was that the students did much more poorly than we expected" ([18] p. 132). The results of this study in this aspect are different. Most working group members knew what to expect. It should be noted that in the original study the expectation were not empirically measured. Nonetheless, the fact that in this experiment four out of six felt that the results, whether poor or high, matched their expectations from the students, imply that the teachers' expectation were more attuned to their students. We cannot rule out, however, the explanation that the teachers, especially in this group, are familiar with previous studies reported on students' behavior, and have colored their expectations accordingly. It may be that the longest-lasting effect of the original MWG has been to depress teachers' expectations of their students' ability!

11. ACKNOWLEDGEMENTS

The working group would like to acknowledge and thank the following colleagues who provided access to their students and helped collect data: Kerttu Pollari-Malmi, Satu Alaoutinen, Timi Seppälä and Teemu Sirkiä; Tomasz Misztur; Luis Fernando de Mingo Lopez, Nuria Gomez Blas, Irina Illina, Bernard Mangeol, Paolo Boldi, Walter Cazzola, Dario Malchiodi, Marisa Maximiano,Vitor Távora, Grzegorz Filo, Pawel Lempa, Lya van der Kamp, Eddy de Rooij, Markku Karhu, Olli Hämäläinen, Lucas Cosson, Erja Nikunen and Antti Salopuro.

12. REFERENCES

[1] Barnes, D. and Kölling, M. *Objects first with Java : a practical introduction using BlueJ.* Pearson Education, Upper Saddle River, N.J., 2011.

[2] Beck, K. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[3] Ben-David Kolikant, Y. (Some) grand challenges of computer science education in the digital age: a socio-cultural perspective. In Anonymous *Proceedings of the 7th Workshop in Primary and Secondary Computing Education.* ACM, , 2012, 86-89.

[4] Bennedsen, J. and Caspersen, M. E. Programming in context: a model-first approach to CS1. In Anonymous *Proceedings of the 35th SIGCSE technical symposium on Computer science education.* (Norfolk, Virginia, USA,). ACM, New York, NY, USA, 2004, 477-481.

[5] Bennett, S., Maton, K. and Kervin, L. The 'digital natives' debate: A critical review of the evidence. British journal of educational technology, 39, 5, 2008, 775-786.

[6] Caspersen, M. E. *Educating novices in the skills of programming.* PhD Thesis, Aarhus University, Science and Technology, Centre for Science Education, Aarhus, Denmark, 2007.

[7] Caspersen, M. E. and Christensen, H. B. Here, there and everywhere - on the recurring use of turtle graphics in CS1. In Anonymous *Proceedings of the Australasian conference on Computing education.* (Melbourne, Australia,). ACM, New York, NY, USA, 2000, 34-40.

[8] Engel, G. and Roberts, E. Computing curricula 2001 computer science. IEEE-CS, ACM.Final Report, 2001.

[9] European Commission. *European Credit Transfer System.* http://ec.europa.eu/education/lifelong-learning-policy/ects_en. htm, 2013, 7/31, 2013.

[10] Fidge, C., Hogan, J. and Lister, R. What vs. How: Comparing Students' Testing and Coding Skills. In Anonymous *Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013)*. Australian Computer Society in the Conferences in Research and Practice in Information Technology (CRPIT), 2013, 97-106.

[11] Ghosh, A. and Kleinberg, J. Incentivizing participation in online forums for education. In *Proceedings of the fourteenth ACM conference on Electronic commerce.* (Philadelphia, Pennsylvania, USA,). ACM, New York, NY, USA, 2013, 525-542.

[12] Guzdial, M. We're too Late for "First" in CS1. Blog@CACM, http://cacm.acm.org/blogs/blog-cacm/102624-were-too-late-for-first-in-cs1, (December 7, 2010).

[13] Hmelo, C. E. and Guzdial, M. Of black and glass boxes: scaffolding for doing and learning. In Anonymous *Proceedings of the 1996 international conference on Learning sciences.* (Evanston, Illinois,). International Society of the Learning Sciences , 1996, 128-134.

[14] Joint ACM/IEEE-CS Task Force on Computing Curricula. Computer Science Curricula 2013: Strawman Draft. http://cs2013.org/strawman-draft/cs2013-strawman.pdf, (February 2012 2012).

[15] Jones, C., Ramanau, R., Cross, S. and Healing, G. Net generation or Digital Natives: Is there a distinct new generation entering university? Computing Education, 54, 3, 2010, 722-732.

[16] Lopez, M., Whalley, J., Robbins, P. and Lister, R. Relationships between reading, tracing and writing skills in introductory programming. In Anonymous *Proceedings of the Fourth international Workshop on Computing Education Research.* (Sydney, Australia,). ACM, New York, NY, USA, 2008, 101-112.

[17] McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K. and Zander, C. Can First-Year Students Program Yet? A Study Revisited. In *ICER '13: Proceedings of the ninth annual international conference on International computing education research.* (Sab Diego, CA, USA, August 2013). ACM, New York, NY, USA, 2013.

[18] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bull, 33, 4, 2001, 125-180. DOI=10.1145/572139.572181.

[19] Rittle-Johnson, B., Siegler, R. S. and Alibali, M. W. Developing conceptual understanding and procedural skill in mathematics: An iterative process. J. Educ. Psychol., 93, 2, 2001, 346.

[20] Rogalski, J. and Samurçay, R. Acquisition of programming knowledge and skills. Psychology of programming, 18, 1990, 157-174.

[21] Sajaniemi, J. and Kuittinen, M. An experiment on using roles of variables in teaching introductory programming. Computer Science Education, 15, 1, 2005, 59-82.

[22] SIGCSE Members. Hand-writing code on exams. http://listserv.acm.org/scripts/wa-ACMLPX.exe?A1=ind1305C&L=SIGCSE-members#7, (15-17 May 2013).

[23] Tew, A. E. *Assessing fundamental introductory computing concept knowledge in a language independent manner.* PhD Thesis, Georgia Institute of Technology, 2010.

[24] Tew, A. E. and Guzdial, M. The FCS1: a language independent assessment of CS1 knowledge. In Anonymous *Proceedings of the 42nd ACM technical symposium on Computer science education.* (Dallas, TX, USA). ACM, New York, NY, USA, 2011, 111-116.

[25] The Joint ACM/AIS/IEEE-CS Task Force on Computing Curricula. Computer Science Curriculum 2008: An Interim Revision of CS 2001.

http://www.acm.org/education/curricula/ComputerScience2008. pdf, 2008.

[26] Van Merriënboer, J. J., Kirschner, P. A. and Kester, L. Taking the load off a learner's mind: Instructional design for complex learning. Educational psychologist, 38, 1, 2003, 5-13.

Appendix A – Clock Task Reference Implementation

Time.java

```
/**
 * Objects of the Time class hold a time value for a
 * European-style 24 hour clock.
 * The value consists of hours, minutes and seconds.
 * The range of the value is 00:00:00 (midnight) to 23:59:59 (one
 * second before midnight).
 *
 * Type your UID here:
 * How long did this take you (hours):
 *
 * @version 1.1
 */
public class Time
{
    // The values of the three parts of the time
    private int hours;
    private int minutes;
    private int seconds;

    /**
     * Constructor for objects of class Time.
     * Creates a new Time object set to 00:00:00.
     * Do not change this constructor.
     */
    public Time()
    {
        this.hours = 0;
        this.minutes = 0;
        this.seconds = 0;
    }

    /**
     * Constructor for objects of class Time.
```

```
 * Creates a new Time object set to h:m:s.
 * Assumes, without checking, that the parameter values are
 * within bounds.
 * For this task, you don't need to worry about invalid parameter values.
 * Do not change this constructor.
 */
public Time(int h, int m, int s)
{
    this.hours = h;
    this.minutes = m;
    this.seconds = s;
}

/**
 * Add one second to the current time.
 * When the seconds value reaches 60, it rolls over to zero.
 * When the seconds roll over to zero, the minutes advance.
 * So 00:00:59 rolls over to 00:01:00.
 * When the minutes reach 60, they roll over and the hours advance.
 * So 00:59:59 rolls over to 01:00:00.
 * When the hours reach 24, they roll over to zero.
 * So 23:59:59 rolls over to 00:00:00.
 */
public void tick()
{
    // Task 1: complete the tick() method
}

/**
 * Compare this time to otherTime.
 * Assumes that both times are in the same day.
 * Returns -1 if this Time is before otherTime.
 * Returns 0 if this Time is the same as otherTime.
 * Returns 1 if this Time is after otherTime.
 */
public int compareTo(Time otherTime)
{
    // Task 2: complete the compareTo method
    return 0;
}

/**
 * Add an offset to this Time.
 * Rolls over the hours, minutes and seconds fields when needed.
 */
public void add(Time offset)
{
    // Task 3: complete the add method
}

/**
 * Subtract an offset from this Time.
 * Rolls over (under?) the hours, minutes and seconds fields when needed.
 */
public void subtract(Time offset)
{
    // Task 4: complete the subtract method
}

/**
 * Return a string representation of this Time.
 * String is of the form hh:mm:ss with always two digits for h, m and s.
 * Do not change this.
```

```
     */
    public String toString()
    {
        return pad(hours) + ":" + pad(minutes) + ":" + pad(seconds);
    }

    /**
     * Returns a string representing the argument value, adding a leading
     * "0" if needed to make it at least two digits long.
     * Do not change this.
     */
    private String pad(int value)
    {
        String sign = "";

        if (value < 0) {
            sign = "-";
            value = -value;
        }

        if (value < 10) {
            return sign + "0" + value;
        } else {
            return sign + value;
        }
    }
}
```

TimeTester.java

```
/**
 * Runs tests on instances of the Time class using the main method of this class.
 * Tests are divided into four sets, one for each of the
 * sub-tasks described in the Time class, which are executed in the
 * order of the sub-tasks.
 * Sets are only attempted if all the previous sets have passed.
 * Tests within a set are attempted even if previous tests in the set have failed.
 *
 * Do not change this class.
 *
 * @author Ian Utting
 * @version 1.1
 */
public class TimeTester
{
    /**
     * Run all of the sets of tests, running each one only if the previous
     * sets have all passed.
     * This makes the results less cluttered if you are attempting the
     * sub-tasks in order.
     */
    public static void main(String [] args)
    {
        if (!tickTests()) return;
        System.out.println("All tick() tests passed.");
        if (!compareToTests()) return;
        System.out.println("All compareTo() tests passed.");
        if (!addTests()) return;
        System.out.println("All add() tests passed.");
        if (!subtractTests()) return;
        System.out.println("All subtract() tests passed.");
        System.out.println("All tests passed.");
    }
```

```
/**
 * Test the tick() method of a Time.
 * All of these tests will run, independent of individual failures.
 */
public static boolean tickTests()
{
    boolean allPassed = true;

    allPassed &= tickTest(new Time( 0, 0, 0), "00:00:01");
    allPassed &= tickTest(new Time( 0, 0,58), "00:00:59");
    allPassed &= tickTest(new Time( 0, 0,59), "00:01:00");
    allPassed &= tickTest(new Time( 0,58,59), "00:59:00");
    allPassed &= tickTest(new Time(00,59,59), "01:00:00");
    allPassed &= tickTest(new Time(23,59,59), "00:00:00");

    Time t = new Time(0, 0, 0);
    allPassed &= tickTest(t, "00:00:01");
    allPassed &= tickTest(t, "00:00:02"); // Same t, ticked twice

    return allPassed;
}

/**
 * Test the compareTo() method of a Time.
 * All of these tests will run, independent of individual failures.
 */
public static boolean compareToTests()
{
    boolean allPassed = true;

    Time t1 = new Time(0, 0, 4);
    Time t1Clone = new Time(0, 0, 4);

    allPassed &= compareToTest(t1, t1, 0);
    allPassed &= compareToTest(t1, t1Clone, 0);

    Time t2 = new Time(0, 0, 5);

    allPassed &= compareToTest(t1, t2, -1);
    allPassed &= compareToTest(t2, t1, 1);

    allPassed &= compareToTest(new Time(2, 2, 2), new Time(1, 2, 2), 1);
    allPassed &= compareToTest(new Time(2, 2, 2), new Time(2, 1, 2), 1);
    allPassed &= compareToTest(new Time(2, 2, 2), new Time(2, 2, 1), 1);
    allPassed &= compareToTest(new Time(1, 2, 2), new Time(2, 2, 2), -1);
    allPassed &= compareToTest(new Time(2, 1, 2), new Time(2, 2, 2), -1);
    allPassed &= compareToTest(new Time(2, 2, 1), new Time(2, 2, 2), -1);

    return allPassed;
}

/**
 * Test the add() method of a Time.
 * All of these tests will run, independent of individual failures.
 */
public static boolean addTests()
{
    boolean allPassed = true;

    allPassed &= addTest(new Time(1, 1, 1), new Time(2, 2, 2), "03:03:03");
    allPassed &= addTest(new Time(0, 0, 59), new Time(0, 0, 1), "00:01:00");
    allPassed &= addTest(new Time(0, 59, 0), new Time(0, 0, 1), "00:59:01");
```

```
    allPassed &= addTest(new Time(0, 59, 59), new Time(0, 0, 1), "01:00:00");
    allPassed &= addTest(new Time(23, 0, 0), new Time(1, 0, 0), "00:00:00");
    allPassed &= addTest(new Time(23, 59, 0), new Time(0, 1, 0), "00:00:00");
    allPassed &= addTest(new Time(23, 59, 59), new Time(0, 0, 1), "00:00:00");
    allPassed &= addTest(new Time(23, 59, 59), new Time(23, 59, 59), "23:59:58");

    return allPassed;
}

/**
 * Test the subtract() method of a Time.
 * All of these tests will run, independent of individual failures.
 */
public static boolean subtractTests()
{
    boolean allPassed = true;

    allPassed &= subtractTest(new Time(2, 2, 2), new Time(1, 1, 1), "01:01:01");
    allPassed &= subtractTest(new Time(0, 1, 0), new Time(0, 0, 1), "00:00:59");
    allPassed &= subtractTest(new Time(1, 0, 0), new Time(0, 1, 0), "00:59:00");
    allPassed &= subtractTest(new Time(1, 0, 0), new Time(0, 0, 1), "00:59:59");
    allPassed &= subtractTest(new Time(1, 1, 1), new Time(1, 1, 1), "00:00:00");
    allPassed &= subtractTest(new Time(1, 1, 1), new Time(0, 0, 2), "01:00:59");
    allPassed &= subtractTest(new Time(1, 1, 1), new Time(0, 2, 2), "00:58:59");
    allPassed &= subtractTest(new Time(1, 1, 1), new Time(2, 2, 2), "22:58:59");

    return allPassed;
}

/**
 * Implementation of an individual tick test.
 */
private static boolean tickTest(Time t, String expected)
{
    String orig = t.toString();

    t.tick();

    if (t.toString().equals(expected)) return true;

    System.out.println("Test: with Time " + orig + ", tick() failed. " +
        "Expected \"" + expected + "\", got \""+ t + "\"");
    return false;
}

/**
 * Implementation of an individual comparison test.
 */
private static boolean compareToTest(Time t1, Time t2, int expected)
{
    int result = t1.compareTo(t2);

    if (result == expected) return true;

    System.out.println("Test: with Time " + t1 + ", compareTo(" + t2 + ") failed. " +
        "Expected \"" + expected + "\", got \"" + result + "\"");
    return false;
}

/**
 * Implementation of an individual addition test.
 */
private static boolean addTest(Time t1, Time t2, String expected)
```

```
    {
        String hdr = "Test: with Time " + t1 + ", add(" + t2 + ") failed. ";
        String origT2 = t2.toString();

        t1.add(t2);

        if (!t2.toString().equals(origT2)) {
            // Second parameter should not be changed
            System.out.println(hdr +
                "Parameter changed from \"" + origT2 + "\"to \""+ t2 + "\"");
            return false;
        }

        if (!t1.toString().equals(expected))
        {
            System.out.println(hdr +
                "Expected \"" + expected + "\", got \""+ t1 + "\"");
            return false;
        }
        return true;
    }

    /**
     * Implementation of an individual subtraction test.
     */
    private static boolean subtractTest(Time t1, Time t2, String expected)
    {
        String hdr = "Test: with Time " + t1 + ", subtract(" + t2 + ") failed. ";
        String origT2 = t2.toString();

        t1.subtract(t2);

        if (!t2.toString().equals(origT2)) {
            // Second parameter should not be changed
            System.out.println(hdr +
                "Parameter changed from \"" + origT2 + "\"to \""+ t2 + "\"");
            return false;
        }

        if (!t1.toString().equals(expected))
        {
            System.out.println(hdr +
                "Expected \"" + expected + "\", got \""+ t1 + "\"");
            return false;
        }
        return true;
    }
}
```