Novice Programmers and the Problem Description Effect

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

Ellie Lovellette
Southern Illinois University Edwardsville
Edwardsville, IL, USA
elovell@siue.edu

John Matta
Southern Illinois University Edwardsville
Edwardsville, IL, USA
jmatta@siue.edu

Bedour Alshaigy
Oxford Brookes University
Oxford, England
bedour.alshaigy-2012@brookes.ac.uk

Brett A. Becker
University College Dublin
Belfield, Dublin 4, Ireland
brett.becker@ucd.ie

Michelle Craig
University of Toronto
Toronto, Ontario, Canada
mcraig@cs.toronto.edu

Jana Jackova
Matej Bel University
Banska Bystrica, Slovakia
jana.jackova@umb.sk

Robert McCartney
University of Connecticut
Storrs, CT, USA
robert@engr.uconn.edu

Kate Sanders
Rhode Island College
Providence, RI, USA
ksanders@ric.edu

Mark Zarb
Robert Gordon University
Aberdeen, Scotland
m.zarb@rgu.ac.uk

ABSTRACT

It is often debated whether a problem presented in a straightforward minimalist fashion is better, or worse, for learning than the same problem presented with a "real-life" or "concrete" context. The presentation, contextualization, or "problem description" has been well studied over several decades in disciplines such as mathematics education and psychology; however, little has been published in the field of computing education. In psychology it has been found that not only the presence of context, but the type of context can have dramatic results on problem success. In mathematics education it has been demonstrated that there are non-mathematical factors in problem presentation that can affect success in solving the problem and learning. The contextual background of a problem can also impact cognitive load, which should be considered when evaluating the effects of context. Further, it has been found that regarding cognitive load, computer science has unique characteristics compared to other disciplines, with the consequence that results from other disciplines may not apply to computer science, thus requiring investigation within computer science.

ITiCSE '16 Working Group Reports, July 09-13, 2016, Arequipa, Peru

DOI: http://dx.doi.org/10.1145/3024906.3024912

This paper presents a multi-national, multi-institutional study of the effects of problem contextualization on novice programmer success in a typical CS1 exercise.

Categories and Subject Descriptors

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

Keywords

context, novice programmers, CS1

1. INTRODUCTION

Elliot Soloway, one of the earliest computing education researchers, tested novice programmers' ability to problem solve and write programs using the Rainfall Problem [48]. The Rainfall Problem asks students to write a program that calculates the average of a set of daily rainfall measurements (hence the name of the problem). The Rainfall Problem is still attractive to programming instructors as a novice programming assignment because it is well written, and seems to be of appropriate difficulty. Nevertheless, as discussed in Section 2, Soloway's experimental results show that few students were able to successfully write a program to solve this problem.

One author of this paper used the Rainfall Problem in an introductory course. One particular detail of the Rainfall Problem stumped a novice programmer, who asked: "What does a daily rainfall value of zero represent?" The answer to the question is that it is a day without rainfall. However awkwardly, this student was trying to make a connection between the numerical problem and the context. Had it not been for the problem description, the student wouldn't have

questioned a value of zero. That question inspired this investigation into the role of context, or "problem description" in the success of novice programmers.

On one hand, context might be motivating to students. Computing an average that corresponds to something real is likely more interesting than just crunching the numbers. On the other hand, understanding the background story, determining which details are relevant and which are not, and what the implications are (for example, that a rainfall amount is never negative), all impose cognitive load on novices, particularly if the language of instruction is not their first language.

Novice programmers are challenged by the combination of problem understanding, finding a solution to the problem, expressing the solution in a programming language, and the environment (development tools, operating system, etc.) in which this is all taking place. The description of the problem could be a significant factor in student success, or failure.

The research question posed here is:

> Does providing context for a programming assignment help novice programmers to solve it?

where "novice programmers" means students in a CS1 class, and "context" means more than a minimal description of input and output – some kind of "real-world" story.

Note the research focus is on the design of individual programming assignments. Adding context and real-world applications to the design of an entire CS1 course (for example, the media computation approach [22]) raises much larger issues such as effect on drop-out rates, effect on underrepresented groups, etc. These are interesting and important questions, but outside the scope of this work. Moreover, with the research question isolated from the course design, the result could have broader implications for computing instructors; that is, an instructor can choose to use real world problem descriptions in assignments relatively independently of other course design issues.

Section 2 presents earlier work on context, from both inside and outside of computing education research. Sections 3 and 4 give the details of this experiment and how the data are analyzed. Sections 5 and 6 describe and discuss the results of the experiment. Finally, Section 7 presents conclusions and suggestions for future work.

2. RELATED WORK

A significant number of publications in various disciplines address the question of context; some research indicates context is helpful, and some points in the opposite direction. A summary of the closely related literature follows.

2.1 Psychology: the Wason Selection Task

The psychology literature suggests that context helps (or in their terms, there is a "thematic facilitation effect"). A well-known problem illustrating this effect, the Wason Selection Task, is the most intensively researched problem in the history of the psychology of reasoning [15, 20, 50]. In the original Wason study, participants were given a no-context version:

> [T]he subject might be presented with four cards showing, respectively, A, 4, D, and 7, and the rule "If there is a vowel on one side of a card, then there is an even number on the other side". The subject is told that each card has a letter on one side and a number on the other side and asked to select just those cards which it is necessary to turn over in order to find out whether the rule is true or false.

The answer is the card with "A" on it – to see if there's an even number on the other side – and the card with "7" on it – to make sure there is *not* a vowel on the other side. In this version of the problem, which has been replicated numerous times, typically less than 10% of the participants give the correct answer [20, p. 407]. This is consistent with work in computing education research that has found that even students who have completed a digital logic course still have misconceptions about logical implication [23].

Later replications added different kinds of context to the Wason rules, for example: "If it rains, the street is wet"; "If I travel to Manchester, I go by car"; "If a letter is sealed, then it has a 50 lire stamp on it"; "If there is L.B.MILL on one side of the envelope, then there is PRINTED PAPER REDUCED RATE ON THE OTHER SIDE"; and "If a person is drinking legally, then the person must be over 19 years of age". [20] Some of these studies found that context made a significant difference, some studies found a small difference, and some found none. Some replication studies arrived at non-confirmatory results (for example, traveling to Manchester).

Griggs and Cox [20] hypothesized that context works *if* it's related to the participant's own experience: "Performance on the selection task is significantly facilitated when the presentation of the task allows the subject to recall past experience with the content of the problem, the relationship expressed, and a counter-example to the rule governing the relationship." [20, p. 417] Thus, a completely arbitrary rule like "If someone is wearing purple, they must be over 18" would not help; a rule such as "If it's winter, it's cold out" would be most helpful for participants who have lived in a place where it's generally true but not always; and a rule about the drinking age would help the large majority of college students. (In the United States, where much of this experiment was run, the drinking age is generally 21, so some students can drink legally and some cannot.) Griggs and Cox ran a carefully-designed experiment with three different rules, including the drinking-age rule, that convincingly supported their hypothesis.

2.2 Education: Cognitive Load Theory

Cognitive load can be defined as "the load imposed on an individual's working memory by a particular (learning) task" [40]. The widely accepted assumption is that working memory only possesses limited capacity and can only deal with a very limited number of information elements at a time [34]. For novices of a domain who have not committed knowledge to long-term memory, working memory is used for fundamental knowledge, reasoning, and problem solving. Thus, it is especially important that learning materials do not overload the available working memory. According to Cognitive Load Theory there are three components of cognitive load - intrinsic, extraneous, and germane [52, 53]. Intrinsic cognitive load is determined by the inherent nature of the learning tasks themselves, while extraneous cognitive load is determined by the manner in which the tasks are presented.

Intrinsic cognitive load cannot be altered by instructional interventions because it is determined by the interaction between the nature of the materials being learned, and the expertise of the learner [53]. Extraneous cognitive load, on the other hand, depends on the format of instruction and can be manipulated with the appropriate instructional design [4]. When designing instructional material, care should be given to eliminate any possible extraneous load [28]. Germane cognitive load is conscious processing using domain schemas guided by instructional design that can take place when both intrinsic and extraneous cognitive load levels are low enough so that some working memory capacity remains unused [52].

Adding context gives students learning to program more to read, requires them to abstract the relevant information from the problem description, and then requires them to translate the relevant information into code. This is a necessary skill for professional programmers, but for novices it may be "extraneous cognitive load".

Computer science has a significant intrinsic cognitive load [19, 35], related to the programming languages used and to understanding and controlling an external computational agent. This kind of problem does not occur frequently in other disciplines, but occurs from the beginning in computer science. As a result, it may be even more important to avoid extraneous cognitive load.

On the other hand, the combination of this intrinsic load and the difference between computer science and other disciplines means that findings from studies in other disciplines might not apply in computer science [35, 36]. Therefore, studies within computing education are warranted [35].

2.3 Math Education: word problems

Mathematical word problems are "presented in the form of a short narrative rather than in mathematical notation" [9]. Compared to numerical computation problems, word problems have "context" in the sense used in this paper.

Word problems are notoriously difficult for students [8]. Language complexity, the use of unfamiliar words, and even the placement of the question (before or after the narrative) all contribute to that difficulty [9]. Wording and semantic factors have been demonstrated to heavily influence problem difficulty. Wording factors refer to the linguistic surface or presentational structure of problems [42]. Even minor variations in the use of specific linguistic elements may have a significant impact on problem difficulty that is not explained by the logico-mathematical structures [41]. Semantic factors include non-mathematical action or situational structure underlying word problems. Studies have shown that problem difficulty varies with this structure which can be static or dynamic, realistic or artificial, intuitively meaningful or more abstract, and familiar or unfamiliar [42].

2.4 Computing Education: context

In computing education, many have argued that context helps. For this purpose, Guzdial [21, p.4] defines "context" broadly, as "the use of a consistent application or domain area, which effectively covers the core areas of a computer science course, provides a source for explanations and a basis for student projects". As examples, he cites media computation and robotics. Esper's magical "code spells" metaphor might also be an example, if used throughout a course [14].

A variety of reasons are given in support of context, including student motivation [12, 18]; enabling students to engage with subjects and concepts they may otherwise have found uninteresting [54]; allowing students to better perceive the relevancy of the material being taught [21, 24, 43]; recruitment and retention of women [43]; retention more generally [21]; connecting students with practitioners [12]; showing students their efforts can positively impact a broader user community [3]; and enabling students to explore broader social and ethical issues [12].

Guzdial [21] has argued that both contextualized and decontextualized courses have value. Contextualized courses help with retention, but students, particularly weak students, can be distracted by the contextual information. Further, de-contextualized information is important for transfer. Few novices can perceive the underlying structural similarity between two problems presented with different contexts. Students eventually need de-contextualized material, except in the unlikely event they can count on spending their whole careers working in the same domain. Thus, Guzdial argued for a contextualized CS1, followed by later de-contextualized courses.

This is a broader definition of context than the one used here, but it is quite plausible that some of these arguments, particularly those involving student motivation, also apply at the level of an individual assignment. Esper et al. [14] presented results on some individual contextualized assignments using CodeSpells, an IDE designed for Java novices. The CodeSpells programs allow students to write "spells" – Java methods that allow the programmer's avatar to do magical things such as levitate or set things on fire. This is a more immersive form of context than simply changing the wording of an assignment. In a small pilot study, students showed high motivation and notable achievement in a 45-minute lesson.

The arguments for and against context raised by Guzdial are echoed in the response to Lister et al. [29]. This paper addressed the question of whether students can read code. Like this study, Lister et al.'s project involved specific questions, but all of their questions were deliberately decontextualized. For example, students were asked to read the following code and identify the contents of array `x` after execution.

```
int[] x = {0, 1, 2, 3};
int temp;
int i = 0;
int j = x.length - 1;

while(i < j)
{
   temp = x[i];
   x[i] = x[j];
   x[j] = 2*temp;
   i++;
   j--;
}
```

The author(s) of this code avoided using meaningful variable names, etc., in order to control for linguistic issues and focus narrowly on students' abilities to read and trace code.

Parsons et al. [37] argue that students' failure to correctly answer questions such as this one is not the students' fault:

"Not being able to answer this question doesn't necessarily demonstrate that one is unable to write code to iterate over and modify the contents of an array, it merely indicates that this abstract and tortuous piece of code is difficult to understand." [37, p. 120] They did not address the issue of transfer, however, or the possibility that students might be distracted or hindered by the added difficulty of processing a program's context.

McCracken et al. [33] raised the question of whether context might make a difference to an individual programming assignment. They asked over 200 students from four institutions in three different countries to write a simple calculator. While they provided three variations on the calculator task, all had context.

After finding that the students did much worse than expected, they conjectured that the context might be a partial explanation [33, p. 136]:

> The specifications of the exercises in this study included details that were not relevant to the solution, which made it difficult for many students to achieve the first learning objective in our framework (abstracting the problem from the description)... [M]any students [...] did not get seem to get [sic] past that point in the problem-solving process.

2.5 Computing Education: Rainfall

Rather than looking at context, Soloway wanted students to be able to see through the context and identify problems with a similar structure. He focused on the underlying patterns in code and how students could learn to recognize and use them.

Soloway viewed student learning through an idea rooted in cognitive theory [31]. In cognitive theory a schema is a chunk of information that has been successfully learned and is contained in memory. Essentially, it is basic knowledge that a student or programmer can recall and use in a constructive way. Once attained by the learner, a schema acts as a single information element in working memory incorporating symbols and relations between them [38]. Recognizing a schema thus temporarily eliminates the need for immediate understanding of the context. Processing a schema actively reduces intrinsic cognitive load and allows the application of relevant knowledge across contexts. In order to recognize a schema, the learner must first be able to abstract it from worked examples, resulting in germane cognitive load which has to be kept low to ensure success [19]. As computer science has inherently high and hard to reduce intrinsic cognitive load, care should be taken to not additionally overburden novices' working memory. Schemas committed to long term memory by appropriate and timely application of germane cognitive load during instruction with carefully chosen materials can result in reduction of extraneous cognitive load. Enhancing schema construction in instructional design reduces learners' overall cognitive load [4].

Soloway's schemas are called plans and goals. A programmer's job is to know how to combine the appropriate plans and goals into working programs. For example, students who learned the plan for finding the average of a generic list of numbers could apply that plan to any problem in which one of the goals is to find the average of some numbers regardless of the problem's context.

In an empirical study, Soloway et al. [49, p.854] explored the relationship between the choices and decisions that novices make to solve a problem and programming language structure. Specifically, they examined the looping strategies that students employ in code submissions. The study involved asking the students to solve the Averaging Problem:

> Write a program that repeatedly reads in integers, until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, it should not count the final 99999.

Although this problem seemed to be easy, students did "surprisingly poorly on this and related problems" [49, p. 854].

Soloway later introduced a version of the Averaging Problem with context:

> Read in integers that represent daily rainfall, and print out the average daily rainfall; if the input value of rainfall is less than zero, prompt the user for a new rainfall. [48, p. 853]

This was a simple version of what has come to be known as the Rainfall Problem, simpler than the one used in this work.

Ebrahimi [13] conducted an experiment in which he, like Soloway, looked at the way students combine plans to solve problems. In his experiment, he used a more complicated version of Soloway's Rainfall Problem:

> Write a program that will read the amount of rainfall for each day. A negative value of rainfall should be rejected, since this is invalid and inadmissible. The program should print out the number of valid recorded days, the number of rainy days, the rainfall average over the period, and the maximum amount of rain that fell on one day. Use a sentinel value of 9999 to terminate the program. [13, p. 470].

This is essentially the same as the first task studied in this work (the Satellite Program, see Section 3.1): it asks for the number of positive data items (the rainy days), the average, and the maximum.

Seppälä et al. [45] provided a useful survey of previous Rainfall Problem literature. Recent projects have used variants of the Rainfall Problem in a functional programming language [17], in C# [47], and examined the role of test writing in solving the problem [26]. However, none of these have focused on the role of context in solving the problem.

2.6 Computing Education: problem-solving

Novices lack many rudimentary problem-solving strategies needed to write programs, like the identification of basic problem elements, the relationship between them, and the steps necessary to construct a solution (appropriately representing information describing the problem and designing algorithms to transform that information from one representation to another). When successful in solving one problem, they frequently fail to apply the same solution techniques to similar problems [46]. These weaknesses are commonly recognized as one of the barriers or difficulties students face while learning to program [2, 25, 44].

Lavonen et al. [27] argue that context helps students with problem-solving: when students create solutions while solving real-world problems, the products created during the

development process become original and appropriate. In light of these findings, several institutions are trying to instill and develop these skills in students implicitly [1, 10, 16] by requiring them to observe "worked examples" and apply them to new but similar problems.

However, the context associated with worked examples may also cause difficulties. Context increases the cognitive load on students, as learners must comprehend all of the secondary information described in the context even if it is not essential to the main problem [51]. This drawback has recently led to a line of work that revisits Soloway's idea of teaching the underlying subgoals and task structure of programs [5, 6, 7, 11, 30, 36, 39].

3. METHODOLOGY

Given the lack of consensus across scientific domains regarding the effect of problem context on performance and the possible non-transferability of results to the field of Computer Science education, we arrive at the research question of whether the context of a programming assignment helps students solve the assignment. We thus postulate the null hypothesis: the contextualization of an individual programming task assignment in a non-themed course will have no significant impact on the performance of novice programmers. The alternate hypothesis is that the context of the programming problem assignment has an effect on students' performance. Our hypothesis testing, therefore, will be based on two-tailed tests, comparing the alternatives of no effect versus some effect: either the mean (or median) performance measures are equal, or they are not equal. This section describes the experiment used to test the hypothesis.

3.1 The Two Tasks

In order to determine the differences in student performance that might result from contextualization of a problem, two tasks were designed. The first task, the Satellite Problem, is essentially Soloway's Rainfall Problem expressed in the context of astronomy. This task requires students to accept a series of inputs, reject negative numbers, stop input on a sentinel value, and report five values – the count of satellites, the sum of all satellites, the average satellites per planet, the maximum number of satellites orbiting a planet, and the number of planets with at least one satellite. The exact English wording of the Satellite Problem is shown in Figure 1.

The Satellite Problem is different from the simple version of Soloway's Rainfall Problem in significant ways. First, it requires the student to calculate more than just the average rainfall. Similar to Ebrahimi's Plan Composition experiment [13], this problem also asks for the maximum value input and the number of values that are greater than zero, which requires the student to collect more information – and combine more goals and plans (using Soloway's terminology) while designing a solution. Second, the Rainfall Problem measures rainfall over a month, which implies the number of rainfall values will never exceed 31 (the maximum number of days in a month). As a result, a student could solve the problem using an array of size 31, resulting in a less general solution. The Satellite Problem doesn't present a limit on the length of the list of input data values. Third, the Rainfall Problem is open to different interpretations: for example, the amount of rainfall per hour, instead of per day. The number of satellites orbiting a planet has an unambiguous

> You are writing a program for use by an astronomer studying a distant galaxy. The astronomer will look through a telescope at each planet in a solar system, and will use the keyboard to enter the number of natural satellites (or moons) that are observed orbiting that planet. The number of satellites is an integer and cannot be negative, although it can be zero. If a negative number is entered, your program should display an error message and ask for a new number. When the astronomer has finished entering the data for a solar system, she will signify the end of the data by entering 9999. At this point, your program should print its results. The number 9999 should not be included in the calculations. When your program finishes, it should print the following results to the screen:
>
> (a) The total number of planets in the solar system
>
> (b) The total number of satellites observed (for all planets)
>
> (c) Average number of satellites per planet
>
> (d) The number of satellites orbiting the planet with the most satellites
>
> (e) The number of planets that have at least one satellite.

Figure 1: The Satellite Problem.

meaning. Fourth, the number of satellites is always an integer, unlike the measurements in the Rainfall Problem, which were all floating-point numbers. Thus, our students, unlike those attempting the Rainfall Problem, needed to avoid integer division errors.

The second task, as shown in Figure 2, is identical to the Satellite Problem, except that all references to context – astronomy, satellites, etc. – have been stripped out. This task is referred to as the "Just the Numbers" Problem (JTN).

Like Soloway's original problem, the instructions for the tasks in this experiment included hints for the participants. Soloway provided basic loop code; the two tasks of this experiment included sample *while*, *foreach* and *for* loops, and code for reading data from a keyboard. An example of the hints given to participants programming in Python is shown in Figure 3. Hints for other programming languages were similar. In addition, the Visual Basic participants were given code for the GUI input/output form. They were asked to write a separate file that contained the program data and logic, which was comparable to the work asked of the C, Java, and Python participants.

In addition to instructions and hints, the tasks included two sample runs of the program. The sample runs provided the desired output of the program for two example input datasets; one in which all of the values entered were positive, and a case where two negative numbers were entered along with positive values. The output examples in English are shown in Figure 4.

Students from six different institutions in four countries: China, Slovakia, the United Kingdom, and the United States, participated in this experiment. English was the language of instruction in China, the United Kingdom, and the United

Your program should read a succession of positive integers from the keyboard. If a negative number is entered, your program should display an error message and ask for a new number. When 9999 is entered, your program should end and print its results. The number 9999 will signify the end of data entry – this number should not be included in the calculations. When your program finishes, it should print the following results to the screen:

(a) How many numbers have been read

(b) The sum of the numbers that have been read

(c) The average value of the numbers that have been read

(d) The largest value that was read

(e) The number of values that are greater than zero

Figure 2: The Just the Numbers Problem.

Hints:
Using input() to read from the keyboard:

```
name = input("Please enter name:  ")
age = int(input("Please enter age:  "))
```

Loops in Python:

```
while(condition):
  # do something

for w in word:
  # iterate over word

for i in range(5):
  # do something 5 times
```

Figure 3: Hints given to participants programming in Python.

States. American and British students represent about 60% of participants, Chinese students receiving instruction in English represent another 26%. In Slovakia, the language of instruction was Slovakian. The instructions for the experiment were given in the language of instruction: the tasks shown here in English in Figure 1 and Figure 2 were translated into Slovakian for the Slovakian participants.

The Flesch-Kincaid Grade Level score for the JTN Problem description is 5.3, representing a U.S. Grade 5 level of reading difficulty. The corresponding score for the Satellite Problem description is 6.8 due, in part, to the presence of multi-syllable words like *astronomer* and *satellite*. The Satellite Problem description contained words used to describe the context not necessary in the JTN Problem description. Because of this, the Satellite Problem description contained 235 words, while the JTN Problem description contained only 167. The Satellite Problem description is therefore (68 words, 41%) longer and more difficult to read than the JTN Problem description, but the reading ability required in both cases should be well within the abilities of a collegiate student.

3.2 Protocol and Participants

As detailed in Table 1, students from six different institutions participated in the study, which have been labeled A through F for convenience of reference. The programming language of instruction varied by institution. Institution F gave the task in both Python and Visual Basic to two different sets of participants. Participants at institutions A to E were given the instructions of their appropriate task on paper and all copies of the paper instructions were collected after the session. Participants from institution F received the instructions online. Participants were not allowed to use any outside resources such as the text, notes, previous lab assignment solutions, or the Internet. They were not allowed to discuss the problem with each other.

Submitted programs were collected from participants electronically and anonymized before analysis. Each researcher obtained the appropriate Institutional Review Board approval from his or her institution, as required.

Table 1 gives the number of participants in each treatment group for each institution. It also includes the maximum time allowed for the task, the percentage of women,[1] and whether or not the participants were volunteering to do the programming activity or received course credit for their submission.

3.3 Institution and Course Descriptions

3.3.1 Institution A

Institution A is a fully accredited Master's level public university with a high undergraduate enrollment profile.

Introduction to Computing is a 4 US credit-hour (equivalent to 6.68 ECTS hours - European credit transfer and accumulation system) Java CS1 course taught in English. The course is structured as 150 minutes per week of lectures augmented by a weekly 110 minute lab section, for 15 academic weeks. In the semester in which the experiment was conducted, there were three lecture sections of approximately 40 students each taught by two different instructors. The lecture sections varied in format from extemporaneous lecture, to prepared lectures, to peer instruction with electronic response devices. In most lab sessions, students solved programming problems in pairs using an IDE. Topics were organized in a fundamentals-first or objects-last approach in Java.

The experiment was conducted during the 15th, and last, lab session as an individual assignment. Students seated next to each other were to be given different tasks, with the goal that the two versions would be distributed approximately evenly in each lab section and overall. Students had 110 minutes to complete their assigned task. Teaching assistants supervised, but did not answer questions or give programming help. The finished programs were submitted electronically.

All participants in the experiment were enrolled in the course. 29 (31%) of the 94 participants majored in Computer Science or Computer Engineering, 27 (29%) majored in Electrical or Mechanical Engineering, 31 (33%) hadn't yet declared a major, and the remaining 7 had majors in various other disciplines.

[1]IRB restrictions did not allow the collection of this information from Institution F.

Sample output:

The red text in the examples below indicate user input

Enter number of satellites (9999 to quit): ***5*** Enter number of satellites (9999 to quit): ***15*** Enter number of satellites (9999 to quit): ***3*** Enter number of satellites (9999 to quit): ***2*** Enter number of satellites (9999 to quit): ***14*** Enter number of satellites (9999 to quit): ***7*** Enter number of satellites (9999 to quit): ***9999*** Number of planets in solar system: 6 Total satellites observed: 46 Average satellites per planet: 7.666666666666667 Satellites orbiting planet with most satellites: 15 Number of planets with at least one satellite: 6	Enter number of satellites (9999 to quit): ***-5*** Negative numbers not allowed! Enter number of satellites (9999 to quit): ***10*** Enter number of satellites (9999 to quit): ***3*** Enter number of satellites (9999 to quit): ***2*** Enter number of satellites (9999 to quit): ***0*** Enter number of satellites (9999 to quit): ***0*** Enter number of satellites (9999 to quit): ***-6*** Negative numbers not allowed! Enter number of satellites (9999 to quit): ***1*** Enter number of satellites (9999 to quit): ***9999*** Number of planets in solar system: 6 Total satellites observed: 16 Average satellites per planet: 2.6666666666666665 Satellites orbiting planet with most satellites: 10 Number of planets with at least one satellite: 4

Figure 4: Example Input and Output Given in Satellite Task. (Note: Text originally displayed in red is shown in italics.)

Institution	Country	Volunteer	Programming Language	Time (minutes)	%age of women	Participants		
						Satellite	JTN	Total
A	United States		Java	110	7	46	47	93
B	United Kingdom	✓	Python 3	120	10	9	14	23
C	China		C	90	33	28	39	67
D	Slovakia		C	90	5	10	12	22
E	United Kingdom	✓	Java	110	40	5	5	10
F	United States		Python 3	50		5	5	10
F	United States		Visual Basic	50		2	5	7
Total					17	107	125	232

Table 1: Participants in the Study by Institution

3.3.2 Institution B

Institution B is a new public university offering courses ranging from foundation to postgraduate courses.

Foundations of Computer Programming is an introduction to the design and implementation of programs using a high-level computer programming language (Python). The module lasts for one academic semester and is intended for foundation students enrolled in a one year course designed to improve their general skills in computing, mathematics and information technology. Upon successful completion, the students can progress directly to their choice of an undergraduate degree in Computer Science, Engineering, Digital media Production or Information Technology Management for Business.

Student volunteers were recruited via email. They were randomly divided into two groups and asked to complete a programming task depending on the group they belong to within an allocated time (two hours). The instructions were in English and no further questions or explanations were allowed during the experiment. Most of the participants finished within an hour and the solutions were electronically submitted to the instructors.

3.3.3 Institution C

Institution C offers programs which draw from Electrical and Communications Engineering, Mathematical Science and Statistics, Computer Science and Electronics, and Business.

Introduction to Programming 2 is a 5 ECTS credit course that follows on from the 5-credit course Introduction to Computer Programming 1. The language of instruction is English and the programming language is C. The course is structured as 90 minutes per week of lecture augmented by a weekly 90 minute lab for 16 academic weeks. In the semester in which the experiment was conducted, there was one lecture section of approximately 150 students and 2 lab sections of approximately 75 students each supervised by

graduate students, who assisted students by answering programming questions. The lecture sections consisted of prepared lectures. In each lab session, students were tasked with programming problems which reflected lecture topics.

The experiment was conducted during the last lab session of a 16 week semester as an individual assignment. Teaching assistants supervised, but did not answer questions or give programming help.

All participants in the experiment were enrolled in the course. 28 (36%) of the 77 participants are first-year Software Engineering majors and 49 (64%) are first-year Internet of Things Engineering majors.

3.3.4 Institution D

Institution D is a fully accredited Master's level public university.

Programming 1 is a 6 ECTS credit CS1 course taught in the Faculty of Natural Sciences in the first year of the Applied Computer Science program. The language of instruction is Slovak, the programming language is C. The course includes 80 minutes per week of lecture and a weekly 80-minute lab for 13 academic weeks. In the semester in which the experiment took place, there was one lecture section of approximately 40 students and 2 lab sections of approximately 20 students. The whole course was taught and supervised by one teacher (T1). In each lab session, students were tasked with programming problems and developed programs using an IDE. The experiment was conducted by another teacher (T2) who taught the same students, within the subject "Semestral Project 1". This second course prepares students for academic writing in computer science topics.

For the experiment, task instructions were translated by T2 to the Slovak language and printed. The experiment was conducted during the last lab session as an individual assignment for which students did not receive any kind of help. The finished programs were submitted electronically.

All participants in the experiment were enrolled in the course "Programming 1" (and "Semestral Project 1"). All 22 (100%) of the participants majored in Applied Computer Science.

3.3.5 Institution E

Institution E is a large public university awarding degrees ranging from Bachelor's to PhD level.

Software Design and Development (first-year) and Object Oriented Software Design (second-year) are both 30-credit (15 ECTS) courses which are taught subsequently over the first two years at the School of Computing Science and Digital Media. Each course lasts a year across the Computer Science and Computing Graphics and Animation BSc degrees. The language of instruction is English, with JavaScript used as the introductory language, and Java used as the main programming language.

An e-mail was circulated to students from both modules, asking for voluntary participation for the experiment. Students were invited to an empty lab were they were randomly allocated tasks. Teaching assistants supervised, but did not answer questions or give programming help. The submitted programs were e-mailed to the lecturer in charge.

All participants in the experiment were enrolled in one of the two degrees described above.

3.3.6 Institution F

Institution F is a fully accredited Master's level urban public college.

Data were gathered from two different courses, Algorithmic Thinking and Introduction to Visual Basic in Business, both taught in English. The two courses had two different instructors, both full-time faculty. Neither was a researcher on this project.

Algorithmic Thinking is a 4 US credit-hour Python CS1 course for CS majors in the Faculty of Arts and Sciences, designed for students who have not had any previous programming experience. While it is intended for CS majors, some of the students are Mathematics Education majors, for whom it is a requirement.

The course is structured as 200 minutes per week of lecture, which varies in form during the semester from lecture to group work to lab sessions, at the instructor's discretion. In the lab sessions, students are assigned problems to solve using an IDE. The course lasts 14 academic weeks. In the semester in which the experiment was conducted there was one section of approximately 24 students.

Introduction to Visual Basic in Business is a 3 US credit-hour introductory programming course for Computer and Information Systems majors in the School of Management. The course is structured as 150 minutes per week of lecture and lasts 14 academic weeks. In the semester in which the experiment took place, there was one section of 15 students.

The Python experiment was conducted during the 13th week of classes; the Visual Basic experiment was conducted during the 14th and final week of classes. In each class, half of the students were given one task and half were given the other. Following the requirements of the local Institutional Review Board (IRB), each student was given a consent form outlining the experiment, and data were collected only from those students who consented. Instructors supervised, but did not answer questions. Neither instructor gave programming help.

All participants in the experiment were enrolled in the course from which their data were gathered. The researcher at institution F was not authorized by the local IRB to collect demographic information. It is highly likely, however, that the Visual Basic students intended to major in Computing and Information Systems, and the Python students intended to major in either Computer Science or Mathematics Education.

4. ANALYSIS

The data collected include 246 submitted files and demographic information about their authors (where permitted by the IRB), identified by institution and task version. Of these 246 submissions, 14 were excluded: two where the student submitted the wrong file (one from Institution A and one from Institution D) and 12 for suspected plagiarism (10 from Institution C and two from Institution F).

This left a total of 232 programs in our final dataset. As shown in Table 2, slightly more than half (54.3%) of these programs were Just the Numbers programs, and slightly less than half (45.7%) were Satellite Problem programs. In each group, there were some programs that compiled (the "compiles"), and some that did not (the "non-compiles").

Two different analyses of this dataset were performed. First, the compiles were analyzed by using grey-box test-

Treatment	Compile	Non-Compile	Total
Satellite	91	15 (14.2%)	106
JTN	109	17 (13.5%)	126

Table 2: Numbers of submissions that compile and do not compile by treatment group (Satellite Problem or Just The Numbers)

Analysis 1: Grey-box (Compiles Only)		Analysis 2: White-box (Sample & Non-compiles)	
Successfully reads sequence of integers	1	Reads all data up to sentinel	1
Stops on sentinel	1	Stops on sentinel	1
Sentinel value not included in calculations	1		
Warns against negative numbers	1	Doesn't use negatives and prints warning	1
Prints correct non-negative count	1	Counts at least one (positive, zero, non-negatives) correctly	1
Prints correct count of positives	1	Counts at least two (positive, zero, non-negatives) correctly	1
Prints correct sum	1	Sums values correctly	1
Prints correct average (given sum and non-negative count)	1	Calculates average (correct in terms of sum and non-negative count)	1
Prints correct maximum	1	Calculates maximum correctly	1
Avoids crashing on divide-by-zero	1	Tests for divide-by-zero	1
		Has code to print results	1
Possible points	10	Possible points	10

Table 3: Point assignments for scoring. (Note: italics indicate a black-box test.)

ing: as shown in Table 3, most items were black-box tested, but for some features, white-box testing was used. Second, a sample of 25% of the programs, including both compiles and non-compiles, was tested using a more extensive white-box testing scheme.

These complementary analyses triangulate the results, as in McCracken et al. [33] and McCartney et al. [32] The remainder of this section describes each of these analyses in more detail.

4.1 Analysis 1: Grey-box Testing of Compiles

Analysis 1 used grey-box testing as shown in Table 3. White-box tests were used for successfully reading in a sequence of integers, which would normally involve the creation of a loop; stopping the reading of input when the sentinel value was entered, with (for example) an appropriate loop condition or selection statement; and excluding the sentinel value from the calculations.

Black-box tests were performed using two test sequences. The first test sequence included several positive numbers, a negative number, a zero, and the sentinel value. For this test sequence, the submission received one point each for correctly calculating the number of values entered (excluding the sentinel), their sum, their average, the largest number and the number of values entered that were greater than zero.

In the case of the average, the point was awarded if the program divided the sum by the count regardless of whether or not those two values were themselves correct.

The absence of typecasting to avoid integer division was not considered an error and did not affect the score since the problem is not present in all of the programming languages used in the study.

The second test sequence was an edge case which included only negative numbers and the sentinel. The calculation of the average would run into a division by zero problem, which could cause abnormal termination (in some languages). If the code guarded against division by zero, either implicitly or explicitly, and avoided a crash or invalid result, it was awarded one point.

We continued the analysis in two ways. First, we assigned a score of 0 to the non-compiling solutions and included them in the sample. Second, we re-ran the analysis including only the compiling programs in the data set.

4.2 Analysis 2: White-box testing of a sample

The Grey-box analysis approach, depending on black-box testing, assigned the same zero score to all non-compiling submissions regardless of the completeness or correctness of the submission. In order to be able to consider the correctness of the submissions that did not compile, a strictly white-box analysis methodology was developed to score the structural features in the programs, as in McCartney et al. [32]. The abstract (language independent) structure of a successful solution to these tasks involves iterating over a number of inputs until the sentinel value is read; during the iterations the program skips negative values, and keeps running totals for the sum of values, the number of values read, and the number of zero (alternatively positive) values, as well as a running maximum value. Once the sentinel is read, the program prints out the required outputs, some stored (sum, number read, maximum) and some calculated (average).

Adding some control information, this can be expanded to:

```
loop:  read in number                          (1)
       if (read in sentinel)                   (2)
          exit loop                            (3)
       else if (number < 0)                    (4)
          print error message                  (5)
       else                                    (6)
          count = count + 1                    (7)
          sum = sum + number                   (8)
          if (number > max)                    (9)
              max = number                    (10)
          if (number > 0)                     (11)
              posCount = posCount + 1         (12)
if (count > 0)                                (13)
       average = sum/count                    (14)
       print stats                            (15)
else print error message                      (16)
```

Programs would optimally check errors, particularly when reporting the average and the maximum when the count of numbers read is zero.

Given this structure, one can extract a number of different structural features:

- Code to iterate through all of the data (the loop structure with enclosed read statement (lines 1-12),
- Code that checks for the end of data (lines 2-3),
- Code that processes each data item within the loop, including code to process negative inputs (line 4-5), non-negative inputs (lines 7-10), and positive inputs (lines 11-12),
- Code to check for no data (line 13),
- Code to calculate results (line 14),
- Code to print out the desired stats (line 15), and
- Code to print out "No data" message (line 16).

The arrangement of these features can vary – there may be a read and sentinel test preceding the loop and at the end of the loop, e.g., but the overall features define what needs to be done within the task. These features also describe an alternative structure for this problem: first read all of the data into a data structure, such as an array or list, then process the data items from the data structure.

To capture these features, two researchers agreed on a set of eleven structural elements to tag in each program. They individually tagged a sample of 29 programs from the complete dataset for presence/absence of each attribute, then resolved their differences through discussion; these discussions helped identify and remove ambiguities from the tag definitions, including an identification of near-miss conditions – logically incorrect versions of attributes that should still be accepted.

The resulting list of attributes is as follows:

Reads Input A program has the potential to read in and use all the data entered by the user. A near miss may fail to use one of the data items in calculations, generally the first one. A program that reads data into a fixed size array counts as correct *only* if the array can store at least 400 values.

Stops on Sentinel The program will always recognize the sentinel value (9999) and stop accepting inputs. Near miss if the program tests whether numbers are greater than or equal to 9999 instead of equal to 9999.

Deals with Negatives For every non-sentinel input, the program must check whether it is negative, and if so print a message and exclude it from the computations. Near miss when the program correctly handles individual negative numbers but not more than one in a row.

Counts Zeros The program counts the number of zeroes entered.

Counts Positives The program counts the number of positive inputs entered.

Counts Non-negatives The program counts the number of non-negative inputs entered.

Sums values The program correctly calculates the sum of the non-negative numbers, which requires adding up all of the positive or non-negative inputs.

Calculates average The program correctly calculates the average of the non-negative numbers based on the sum and non-negative count (which do not have to be correct). Near miss if the program does integer division or divides by zero if the sentinel is the first input.

Calculates maximum The program correctly calculates the maximum value entered (excluding the sentinel). Near miss if it reports a maximum of zero for empty sets of numbers.

Prints results Te program displays the results calculated (regardless of whether or not the results themselves are correct) using labels from the problem description. Near miss if the program either does not print or prints label only for values that were not calculated.

Checks for Divide-by-zero The program avoids dividing by zero when the sentinel is the only number entered, and does not attempt to compute the average in this case.

The different count tags correspond to different ways the program might obtain the necessary information for the results: the number of numbers greater than or equal to zero and the number greater than zero. Counting any two of zeroes, positives, or non-negatives is sufficient to obtain the necessary results.

Given these tags, a point assignment was developed to enable comparison of this analysis with the Analysis 1 scoring. One point was given per attribute except for the three counts – for the three counts a maximum of two points were awarded, since the third count can be obtained from the other two. This assignment maps a set of tags to a value between 0 and 10, the same range as the Analysis 1 scores. The Analysis 1 and Analysis 2 schemes are comparable, as shown in Table 3.

After finalizing the white-box tagging scheme, a sample of 58 programs were selected, 25% of the entire dataset, for tagging. The sample was created by selecting from among the randomized file names assigned to submissions when the programs were anonymized. An equal number of Satellite and Just the Numbers programs were chosen from each of the institutions. The tagging was done by three researchers, two for each program; differences were resolved through discussion.

The number of items by institution and problem type in the resulting sample is given in Table 4. The sample also included both compiles and non-compiles: 14 of the 58 programs in the sample were non-compiles, with at least one non-compile from each institution except E (which had no non-compiling submissions).

5. RESULTS

Figure 5 presents the distribution of the grey-box testing scores from Analysis 1. Figure 6 presents the distribution of the white-box testing scores from Analysis 2. Recall that the grey-box testing was applied to the 200 submissions that compiled without error; the 32 non-compiles were given a score of -1, so they are represented by the left-most bars of

Institution	Just the Numbers	Satellite	Sample Size	Population Size
A	8	12	20	93
B	3	4	7	23
C	5	5	10	67
D	5	3	8	22
E	3	3	6	10
F	2	1	3	10
F	3	1	4	7
Total	29	29	58	232

Table 4: Our sample, by program type and by institution.

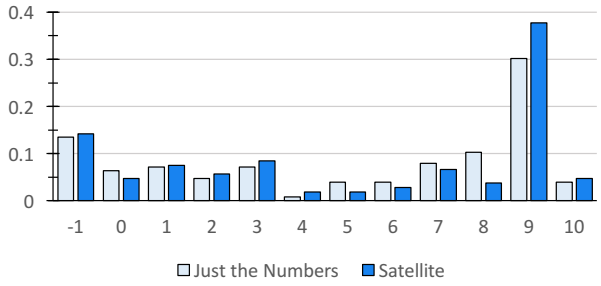

Figure 5: Distribution of Grey-box Testing Scores by Treatment Group. All programs, n = 232.

the graph. The white-box testing was applied to our sample of 25% (58) of the programs; the same test was applied to both compiles and non-compiles. Both distributions are clearly not normal – approximately a third of each group has a score of 9.

In the following sections, we will discuss the analyses done using these data.

5.1 Results 1: Compiled Programs

We compared the distributions of the grey-box scores in two ways. First, non-compiling programs were included with a score of zero. In this case, we ran a t-test on the scores. The mean score for the Satellite treatment group was 5.44 (out of 10) and for the Just the Numbers group was 5.31. A t-test confirms that these are not statistically significantly different ($p = 0.798$).

Second, we ran the same analysis on the compiling submissions only. This gave analogous results with mean scores of 6.33 for Satellite and 6.15 for Just the Numbers, which are not statistically significantly different ($p = 0.707$).

In both cases, the students in the Satellite group performed marginally better, but the difference is not statistically significant. In addition, the score distributions of the two institutions with the most submissions (Institutions A and C) showed no statistically significant difference in performance between the Satellite and the Just the Numbers groups. (See Table 5 for details.)

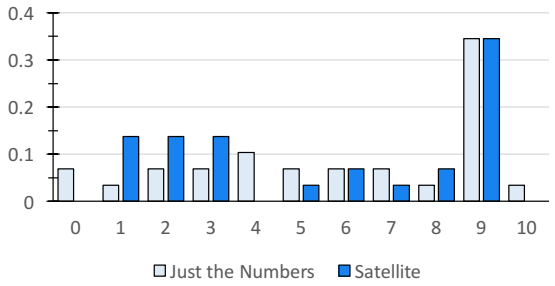

Figure 6: Distribution of White-box Testing Scores by Treatment Group. Sample of programs, n = 58.

5.2 Results 2: White-box Testing of a Sample

The distributions of white-box testing scores for the sample programs look remarkably similar for the Satellite and Just the Numbers submissions, as can be seen in Figure 6. Indeed, in a comparison of the 29 Satellite programs and the 29 Just-The-Numbers programs in the sample, both a two-tailed Mann-Whitney test ($U = 389, p = 0.62$) and t-test ($t(56) = 0.49, p = 0.63$) find no statistically significant differences.

A Shapiro-Wilk test verifies that neither the distribution of the Just the Numbers group nor the distribution of the Satellite group is normal, with a 9.89 variance for Just the Numbers and 10.89 for Satellite.

5.3 Comparing the White-Box and Grey-Box Score Distributions

The grey-box and white-box testing can be used to triangulate the analysis. A comparison of Figures 5 and 6 shows the score distributions are very similar for the two measures. Considering the 58 programs in the 25% sample where both scores exist, the average grey-box score is 4.74, while the average white-box score is 5.83. A significant cause of that difference is due to the 14 non-compiling programs that grey-box analysis scores as zero; their average white-box score is 2.71. The average grey-box and white-box scores for the 44 compiling programs are much closer in value: 6.82 and 6.25, respectively. A one-tailed paired t-test for the compiling programs resulted in an insignificant difference: $t(43) = 1.68, p = 0.091$. Given the non-normality of the distributions, we ran a Wilcoxon paired test on the compiling program scores as well; the results were consistent with the t-test: insignificant difference, $V = 201, p = 0.141$. Not surprisingly, including the non-compiling program scores in this analysis finds a significant difference.

A Pearson's correlation test on the grey-box and white-box scores for the 44 compiling submissions shows the scores are highly correlated ($cor(42) = 0.77, p < 0.001$).

5.4 Results 3: Combining the Analyses

Given the lack of statistically significant difference between the white-box and grey-box testing (as discussed above in Section 5.3), we did a third, combined analysis in which the grey-box scores were used for the compiled programs and the white-box scores were used for the non-compiles. We will call this the *combined score*.

Analyzing the combined score is equivalent to rerunning our original tests from Results 1 except that instead of either omitting or assigning zero to each non-compiling submission,

we use its white-box testing score. The thought was that by assigning a more meaningful score to the non-compiling submissions in the larger data set, we would expose any real difference that was previously masked by the lack of data from the non-compiles.

In order to do this, we required white-box testing (Analysis 2) scores for all of the non-compiles. Using the same process, the three researchers who did the 25% sample also tagged the remaining 18 non-compiling submissions that were not included in that sample.

Again, we found no statistically significant difference between the treatment groups. The means of the combined scores (out of 10) were 5.91 (Just the Numbers) and 5.78 (Satellite). This small difference is not statistically significant ($t(230) = 0.30, p = 0.76$). Given that the data are not normally distributed, it can be argued that a non-parametric test would be more appropriate. A Wilcoxon rank sum also finds the difference not to be statistically significant ($W = 6582, p = 0.83$).

5.5 Results: Gender

Table 1 includes percentages of women within each institution. In total there were 36 females, 175 males and 21 participants for whom the gender was unspecified. Of those who identified as females, 15 were assigned to the Satellite Problem group and 21 were assigned to the Just the Numbers group. A chi-squared test ($\chi = 0.1486, df = 1, p = 0.6999$) confirmed that the random assignment to treatment group was not biased by gender.

Table 5 includes the results comparing performance of only the women and only the men. As with the entire population, no significant differences were found in the performances on the different tasks.

5.6 Results: Proportion of Non-Compiles

The numbers of submissions which did and did not compile for the Satellite and Just the Numbers programs are shown in Table 2. Here again there is no statistically significant difference between the two groups. A chi-squared test ($\chi(1) = 0.021, p = 0.885$) confirms the distributions between the two groups are not statistically significantly different.

5.7 Results: Summary

All of the above analyses agree: there is no statistically significant difference between student performance on the Just the Numbers programs and on the Satellite programs. Thus, we failed to reject the null hypothesis. It is our belief that the data show that novice programmers exhibit equivalent rates of success on contextualized and generic problems.

6. DISCUSSION

6.1 A Multi-Institutional Global Study

Students participating in this study spanned six institutions in four countries from three continents. This strengthens the negative result of the study because the outcome – no difference between the Satellite Problem and Just the Numbers treatment groups – was observed in several different situations: different classrooms, different types of CS1 courses, different types of students and different cultural backgrounds.

6.1.1 Different CS1 Courses and Student Populations

The CS1 courses included in the study differed in many facets including schedule, programming language, and course content – for instance, some of the participating CS1 courses may have introduced more advanced data structures than others. All students were completing a CS1 course, but not all students were enrolled specifically in a Computer Science degree – some were enrolled in related degrees such as Software Engineering, Internet of Things Engineering, Mathematics Education or Digital Media. The motivation for including such a diverse set of institutions and students was to give a representative sample of CS1 courses offered globally, but whether the students and institutions who participated in this study are, in fact, representative of students in general is difficult to determine.

Not only did the majors of the students vary across institutions, their skill level was different as well. Some of the students were true novices and the CS1 course they were enrolled in at the time was their first. Others were in their second year of instruction. It is difficult to find a measure to accurately rank students across institutions. In addition, the courses themselves differed significantly – from the course structure and focus, through the programming language to the credit weight. Also, the gender balance varied drastically, ranging from 5% to 40% for the five institutions reporting (for one institution these statistics could not be reported).

In two institutions students were non-native English speakers. In one of these the language of instruction was English, while in the other the language of instruction was the students' native language. This is discussed in more detail later in this section.

6.1.2 Different computer languages

The observed performance varied across computer languages. In terms of the percent of programs that did not compile, Java (4.9%) was much lower than C (20.2%) and Python (24.2%) (all of the Visual Basic programs compiled, but there were only 7 of them). These differences are hard to isolate to language, however: they also correspond to the institutions that use those languages, and they may be a function of the development environments used. There was no apparent difference for the two tasks.

The difficulty of white-box testing did seem to vary with language as well – the C programs were often more difficult to analyze than the Java or Python programs. It may be that C programs are just harder to read, especially when written by novices who can write correct code that violates convention, such as using a stack to store data, using `push` to store the input data, then accessing the data of the stack as an array when processing the input set. The difficulty of reading the code seemed unrelated to the task, however.

6.1.3 Challenges of a Multi-institutional Study

Multi-institutional, multi-national studies have inherent challenges. While they do have advantages – a larger pool of students from a variety of institutions, making the results potentially more generalizable – they introduce significant problems of coordination and communication. As one of the first multi-national, multi-institutional studies in computing education, McCracken et al. [33] encountered these issues head-on and discussed them in their report. The researchers in their group were allowed to choose any or all of three

Group	Mean Satellite	Mean JTN	Confidence Interval	p	Significant?
All submissions Non-Compiles Scored as Zero	5.44	5.31	(-0.85, 1.11)	0.798	No
All submissions Compiling Only	6.33	6.15	(-0.75, 1.11)	0.707	No
Women submissions Non-Compiles Scored as Zero	5.53	5.52	(-2.61, 2.63)	0.994	No
Women submissions Compiling Only	6.92	6.11	(-1.57, 3.19)	0.488	No
Men submissions Non-Compiles Scored as Zero	5.78	5.67	(-1.00, 1.23)	0.841	No
Men submissions Compiling Only	6.58	6.67	(-1.11, 0.94)	0.866	No
Group A Only Non-Compiles Scored as Zero	6.50	6.66	(-1.52, 1.20)	0.816	No
Group A Only Compiling Only	7.12	6.96	(-1.08, 1.40)	0.794	No
Group C Only Non-Compiles Scored as Zero	6.00	5.23	(-1.10, 2.64)	0.410	No
Group C Only Compiling Only	7.30	7.56	(-1.70, 1.20)	0.729	No

Table 5: Mean scores and confidence intervals for different groups of participants

variations on the calculator problem; some students did the problem voluntarily, some didn't; whether students had help from the instructor, textbooks, or other resources may have varied; there was no centralized approval for translations of the instructions into other languages; and the problem was significantly harder in Java than in other programming languages. (Java did not yet have a `Scanner` class.)

We have tried to avoid these problems as much as possible. As McCracken et al. [33] recommended, we had a single coordinator for the project, the working group leader. The leader had run pilot versions of the experiment before distributing it to the group. All the student programs were anonymized and kept in a single online repository, accessible to all the members of the group. In addition, we took steps to ensure that the problem was comparable in different languages (for example, giving the Visual Basic students the GUI code they needed). All institutions used the same instructions, with only the modifications necessary to translate the instructions into the students' language of instruction or to fit the assignment to a particular programming language. All students completed the task in a supervised setting, without any resources or assistance. We tested the programs for evidence of student collaboration, and where we found suspected plagiarism, we excluded those programs from the data, as previously noted.

Nevertheless, there were still some unavoidable variations. Some students were volunteers; some were not. They wrote the program in the context of different courses, taught by different instructors, using different books, emphasizing different topics, and – due to local constraints – they had different amounts of time to solve the problem. Even though we believed that it would be interesting to capture how much time individual students spent working on their solution, not all of the institutions were able to collect that data. Most of the students did the English-language version of the problem; for many of them, English was not their first language.

6.2 Threats to Validity

While a number of threats to validity such as dependence on a particular institutional situation, are partially mitigated by the inclusion of a number of varied situations, other threats are inherent to the study design itself and present regardless of the classroom setting.

6.2.1 Are students giving their best?

The performance of students on either task can be strongly influenced by their motivation. Some of the students in our population received course credit, while others were volunteers. Regardless of their situation, the question of whether their effort was representative of their actual skill remains. As the experiment took place at the end of the semester, it is possible that students who had already achieved the desired grade in the course did not strain to deliver their best performance. Volunteers, on the other hand, might have lacked the motivation of course credit to do their best.

6.2.2 Are the tasks different enough?

It is possible that the two taskss completed by students were not different enough to affect student performance. Our Just the Numbers task was designed to be as "context-free" as possible without being vague or ill-defined, but a completely context-free task is, in practice, not achievable.

Similarly, our Satellite task was designed to have a "real-world" context, but it is possible that the particular Satellite context we employed was not dissimilar enough from the Just the Numbers task to make student performance significantly different from that on the Just the Numbers task. Both of these threats could be contributors to the lack of significant difference between the submitted solutions.

6.2.3 Is the Satellite context the right choice?

It is also possible that the Satellite context is not of a type that will affect student performance, compared to the Just the Numbers task. As discussed in Section 2.3, wording and semantic factors have been demonstrated to heavily influence problem difficulty of mathematical word problems. Similarly, as discussed in Section 2.1, it has been shown in psychology that context can contribute positively to successfully completing a task if the context is related to the participant's own experience. We expected that all our students would be equally unfamiliar with astronomy. Possibly that lack of familiarity meant that the context did not help our students; or possibly some of them did have some background in astronomy, and entering data about satellites is not an authentic astronomers' task. Authenticity and relatability, and the lack thereof, may affect motivation and performance.

We don't really have a way of classifying context. It would make sense to repeat the experiment with several equivalent tasks wrapped in context drawn from different domains, to test whether the contextualization subject influences novice programmers' performance. It would also be interesting to collect and analyze students' self-reported assessment of their engagement, effort, and ability to relate to a problem, as well as their perception of the difficulty of the task in the form of one or more survey questions. These data would provide a subjective way to measure cognitive load. Data for objective measures can be collected by behavioral observations coupled with the performance on the tasks. Ideally, we would be able to find one, or more, contexts that achieve the thematic facilitation effect of a relatedly contextualized Wason Selection Task that doesn't increase extraneous cognitive load.

6.2.4 Are results affected by non-native English speakers?

Institution C's language of instruction is English, but all of the students are non-native speakers. Some of the students who submitted programs from other institutions were likely also non-native speakers, although we do not have that data.

Conceivably, for non-native English speakers who were being taught in English, increased extraneous cognitive load could have been a factor. We believe that the cognitive load would be higher for the Satellite context – the Flesch-Kincaid Grade Level of the Satellite Problem was 6.8 compared to 5.3 for the Just the Numbers task.

For non-native speakers there would thus be two sources of extraneous cognitive load: the context plus a foreign language. It is possible that this excess cognitive load contributed to non-native English speakers performing worse on the Satellite program. Additionally, the students who are non-native English speakers but being instructed in their native language could also be experiencing a higher cognitive load due to the fact that they were programming in a language with English keywords.

6.3 Implications for Teaching

If our results are correct – that in a CS1 course, problem context does not help students solve a given problem – it would have profound implications for teaching. Developing and marking assessment is time-consuming and developing relevant, interesting, and "real-world" context adds to development overhead. If this overhead has no benefit to learning, CS1 educators could simply not bother with context.

However, context may have benefits beyond simply solving a given problem correctly. For instance we did not specifically measure the effect of context on student motivation or engagement. Obviously these factors may be intertwined with problem success, but measuring engagement and motivation on their own is a different question than the one we sought to answer. Nonetheless, it is possible that context does contribute positively to student motivation and engagement. If this is the case, then CS1 educators would have a reason for adding context to assignments.

Another possible reason that context may be beneficial is its relationship to authentic assessment, which has been shown to have positive learning benefits. If the results of this study hold, and context does not have an effect on students' ability to solve a given problem, more answers on the effects on motivation, engagement and the role of context in authentic assessment need to be sought – in a computer science setting – before CS1 educators decide if including context in their assessments is worthwhile.

Additionally, we only investigated CS1 courses. The role and effect of context in other computer science courses and levels needs to be explored. Finally, the additional cognitive load that context brings (and the fact that this load could be different for different students) needs to be carefully considered if contextual assessments are used by computer science educators.

7. CONCLUSIONS AND FUTURE WORK

The evidence found in this study fails to reject the null hypothesis postulating that the context of a problem leads to no significant difference in novices' success in writing the specific programs we studied. As discussed above, the validity of the results is possibly influenced by a variety of factors. An instructor introducing novices to programming could take this study's outcome to support a decision to limit time and effort spent creating contextualized problems in favor of devoting resources to other aspects of the course. However, as the effects of individual problem context on student motivation and engagement are not fully known, these should be weighed as carefully as possible before deciding not to include context in CS1 problems.

Another interpretation of our results is that while context does not *help* students write better programs, it also does not *hurt.* Thus, if the inclusion of context is found to positively influence student motivation and engagement, then the time spent to contextualize programming tasks may be time well spent.

Replications of this experiment could conceivably shed additional light on our research question. Not all replications of psychology experiments investigating this topic confirmed the prior work. Perhaps more interesting work lies in variations on this experiment. Specifically, it may be possible to construct computing problems that achieve the effect of the Wason Selection Task with a positive impact on programming tasks. Coupled with the potential for increasing student motivation and engagement, finding such a result would provide a compelling reason to use context in individual programming assignments in CS1.

Acknowledgments

Thanks to Walter Gall and Michael Hayden for their help with data collection at a particularly busy time.

8. REFERENCES

[1] T. Beaubouef, R. Lucas, and J. Howatt. The UNLOCK system: Enhancing problem solving skills in CS-1 students. *ACM SIGCSE Bulletin*, 33(2):43–46, 2001.

[2] T. Beaubouef and J. Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *ACM SIGCSE Bulletin*, 37(2):103–106, 2005.

[3] E. Brannock, R. Lutz, and N. Napier. Integrating authentic learning into a software development course: An experience report. In *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*, pages 195–200. ACM, 2013.

[4] R. Brunken, J. L. Plass, and D. Leutner. Direct measurement of cognitive load in multimedia learning. *Educational Psychologist*, 38(1):53–61, 2003.

[5] R. Catrambone. Improving examples to improve transfer to novel problems. *Memory & Cognition*, 22(5):606–615, 1994.

[6] R. Catrambone. Aiding subgoal learning: Effects on transfer. *Journal of Educational Psychology*, 87(1):5, 1995.

[7] R. Catrambone. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General*, 127(4):355, 1998.

[8] D. D. Cummins, W. Kintsch, K. Reusser, and R. Weimer. The role of understanding in solving word problems. *Cognitive Psychology*, 20(4):405–438, 1988.

[9] G. Daroczy, W. M., W. Meurers, and H.-C. Nuerk. Word problems: A review of linguistic and numerical factors contributing to their difficulty. *Frontiers in Psychology*, 6:348, 2015.

[10] M. de Raadt, M. Toleman, and R. Watson. Training strategic problem solvers. *ACM SIGCSE Bulletin*, 36(2):48–51, 2004.

[11] M. De Raadt, M. Toleman, and R. Watson. Incorporating programming strategies explicitly into curricula. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research-Volume 88*, pages 41–52. Australian Computer Society, Inc., 2007.

[12] J. DeWitt and C. Cicalese. Contextual integration: A framework for presenting social, legal, and ethical content across the computer security and information assurance curriculum. In *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development*, pages 30–40. ACM, 2006.

[13] A. Ebrahimi. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies*, 41:457–480, 1994.

[14] S. Esper, S. R. Foster, and W. G. Griswold. Codespells: Embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, pages 249–254. ACM, 2013.

[15] J. S. B. Evans, S. E. Newstead, and R. M. Byrne. *Human reasoning: The psychology of deduction*. Psychology Press, 1993.

[16] C. Fidge and D. Teague. Losing their marbles: Syntax-free programming for assessing problem-solving skills. In *Proceedings of the 11th Australasian Conference on Computing Education-Volume 95*, pages 75–82. Australian Computer Society, Inc., 2009.

[17] K. Fisler. The recurring rainfall problem. In *ICER '14*, 2014.

[18] A. Forte and M. Guzdial. Motivation and nonmajors in computer science: Identifying discrete audiences for introductory courses. *IEEE Transactions on Education*, 48(2):248–253, 2005.

[19] S. Garner. Reducing the cognitive load on novice programmers. In P. Barker and S. Rebelsky, editors, *Proceedings of EdMedia: World Conference on Educational Media and Technology 2002*, pages 578–583, Denver, Colorado, USA, 2002. Association for the Advancement of Computing in Education (AACE).

[20] R. A. Griggs and J. R. Cox. The elusive thematic-materials effect in Wason's selection task. *British Journal of Psychology*, 73(3):407–420, 1982.

[21] M. Guzdial. Does contextualized computing education help? *ACM Inroads*, 1(4):4–6, 2010.

[22] M. Guzdial and E. Soloway. Teaching the Nintendo generation to program. *Communications of the ACM*, 45(4):17–21, Apr. 2002.

[23] G. L. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Proof by incomplete enumeration and other logical misconceptions. In *Proceedings of the 4th International Workshop on Computing Education Research*, ICER '08, pages 59–70, 2008.

[24] N. Herrmann and J. L. Popyack. Creating an authentic learning experience in introductory programming courses. In *ACM SIGCSE Bulletin*, volume 27, pages 199–203. ACM, 1995.

[25] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, volume 37, pages 14–18. ACM, 2005.

[26] A.-J. Lakanen, V. Lappalainen, and V. Isomöttönen. Revisiting rainfall to explore exam questions and performance on cs1. In *In Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 40–49, 2015.

[27] J. M. Lavonen, V. P. Meisalo, and M. Lattu. Problem solving with an icon oriented programming tool: A case study in technology education. 2001.

[28] J. Leppink, F. Paas, T. Van Gog, C. P. van Der Vleuten, and J. J. Van Merrienboer. Effects of pairs of problems and examples on task performance and different types of cognitive load. *Learning and Instruction*, 30:32–42, 2014.

[29] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, et al. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, volume 36, pages 119–150. ACM, 2004.

[30] L. E. Margulieux, M. Guzdial, and R. Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the 9th Annual International Conference on International Computing Education Research*, pages 71–78. ACM, 2012.

[31] R. E. Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys*, 13(1):121–141, 1981.

[32] R. McCartney, J. Boustedt, A. Eckerdal, K. Sanders, and C. Zander. Can first-year students program yet?: A study revisited. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 91–98, New York, NY, USA, 2013. ACM.

[33] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4):125–180, 2001.

[34] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information, 1956. One of the 100 most influential papers in cognitive science: http://cogsci.umn.edu/millennium/final.html.

[35] B. B. Morrison. Computer science is different!: Educational psychology experiments do not reliably replicate in programming domain. In *Proceedings of the 11th Annual International Conference on International Computing Education Research*, pages 267–268. ACM, 2015.

[36] B. B. Morrison, L. E. Margulieux, and M. Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*, pages 21–29. ACM, 2015.

[37] D. Parsons, K. Wood, and P. Haden. What are we doing when we assess programming? In *Proceedings of the 17th Australasian Computing Education Conference*, volume 27, page 30, 2015.

[38] E. Pollock, P. Chandler, and J. Sweller. Assimilating complex information. *Learning and Instruction*, 12(1):61 – 86, 2002.

[39] A. Renkl and R. K. Atkinson. Learning from examples: Fostering self-explanations in computer-based learning environments. *Interactive Learning Environments*, 10(2):105–119, 2002.

[40] A. Renkl and R. K. Atkinson. Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational Psychologist*, 38(1):15–22, 2003.

[41] K. Reusser. Problem solving beyond the logic of things: Contextual effects on understanding and solving word problems. *Instructional Science*, 17(4):309–338, 1988.

[42] K. Reusser. Success and failure in school mathematics: Effects of instruction and school environment. *European Child & Adolescent Psychiatry*, 9(2):S17–S26, 2000.

[43] L. Rich, H. Perry, and M. Guzdial. A CS1 course designed to address interests of women. In *ACM SIGCSE Bulletin*, volume 36, pages 190–194. ACM, 2004.

[44] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[45] O. Seppälä, P. Ihantola, E. Isohanni, J. Sorva, and A. Vihavainen. Do we know how difficult the rainfall problem is? In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 87–96, New York, NY, USA, 2015. ACM.

[46] V. J. Shute. Who is likely to acquire programming skills? *Journal of Educational Computing Research*, 7(1):1–24, 1991.

[47] Simon. Soloway's rainfall problem has become harder. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*, LATICE '13, pages 130–135, Washington, DC, USA, 2013. IEEE Computer Society.

[48] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[49] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.

[50] D. Sperber, F. Cara, and V. Girotto. Relevance theory explains the selection task. *Cognition*, 57(1):31–95, 1995.

[51] J. Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, 22(2):123–138, 2010.

[52] J. Sweller, J. J. G. van Merrienboer, and F. G. W. C. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10(3):251–296, 1998.

[53] J. J. Van Merrienboer and J. Sweller. Cognitive load theory and complex learning: Recent developments and future directions. *Educational Psychology Review*, 17(2):147–177, 2005.

[54] S. Yarosh and M. Guzdial. Narrating data structures: The role of context in CS2. *Journal on Educational Resources in Computing*, 7(4):6, 2008.