Department of Computer Science
Series of Publications A
Report A-2017-4

Retention in Introductory Programming

Arto Hellas

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium VI, Forest House, on October 27th, at noon.

University of Helsinki
Finland

Supervisors
Jaakko Kurhila, Matti Luukkainen and Jukka Paakki
University of Helsinki, Finland

Pre-examiners
Nickolas Falkner, University of Adelaide, Australia
Mikko-Jussi Laakso, University of Turku, Finland

Opponent
Erkki Sutinen, University of Turku, Finland

Custos
Tommi Mikkonen, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911, telefax: +358 9 876 4314

ISSN 1238-8645
ISBN 978-951-51-3797-5 (paperback)
ISBN 978-951-51-3798-2 (PDF)
Computing Reviews (1998) Classification: K.3.2, K.3.1, D.2.8
Helsinki 2017
Unigrafia

Retention in Introductory Programming

Arto Hellas

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
arto.hellas@cs.helsinki.fi

PhD Thesis, Series of Publications A, Report A-2017-4
Helsinki, October 2017, 68 + 88 pages
ISSN 1238-8645
ISBN 978-951-51-3797-5 (paperback)
ISBN 978-951-51-3798-2 (PDF)

Abstract

The introductory programming course is one of the very first courses that computer science students encounter. The course is challenging not only because of the content, but also due to the challenges related to finding a place in a new community. Many have little knowledge of what to expect from university studies, some struggle to adjust their study behavior to match the expected pace, and a few simply cannot attend instruction due to e.g. family or work constraints. As a consequence, a considerable number of students end up failing the course, or pass the course with substandard knowledge. This leads to students failing to proceed in their studies at a desirable pace, to students who struggle with the subsequent courses, and to students who completely drop out from their studies.

This thesis explores the issue of retention in introductory programming courses through multiple viewpoints. We first analyze how the teaching approaches reported in literature affect introductory programming course pass rates. Then, changes on the retention at the University of Helsinki are studied using two separate approaches. The first approach is the use of a contemporary variant of Cognitive Apprenticeship called the Extreme Apprenticeship method, and the second approach is the use of a massive open online course (MOOC) in programming for recruiting students before they enter their university studies. Furthermore, data from an automatic assessment system implemented for the purposes of this thesis is studied to determine how novices write their first lines of code, and what factors contribute to the feeling of difficulty in learning programming.

On average, the teaching approaches described in the literature improve the course pass rates by one third. However, the literature tends to neglect the effect of intervention on the subsequent courses. In both studies at the University of Helsinki, retention improved considerably, and the students on average also fare better in subsequent courses. Finally, the data that has been gathered with the automatic assessment system provides an excellent starting point for future research.

Computing Reviews (1998) Categories and Subject Descriptors:

K.3.2	Computer and Information Science Education
K.3.1	Computer Uses in Education
D.2.8	Software Engineering – Metrics

General Terms:
Design, Experimentation, Human factors

Additional Key Words and Phrases:
cognitive apprenticeship, course material, continuous feedback, instructional design, programming education, best practices, learning by doing, testing, automatic assessment, verification, distance education

Acknowledgements

I am deeply grateful to my family and friends for tolerating my selfish pursuit of my work. While some have argued that selfishness may be a virtue that leads to social progress, there is little virtue in being mentally or physically absent from the lives of those who truly matter the most.

My special thanks go to Matti Paksula and Matti Luukkainen who asked me to take part in redesigning our introductory programming courses. Their question led me to this research. I am also grateful to Jaakko Kurhila and Thomas Vikberg, who have been a crucial part of the early stages of this process. I am deeply grateful to Jukka Paakki and Tommi Mikkonen who chose to champion my work even though it does not belong to the traditionally conducted research at the Department of Computer Science.

Big thanks go to the past and current members and affiliates of the Agile Education Research Group, including Juho Leinonen, Henrik Nygren, Leo Leppänen, Martin Pärtel and Jarmo Isotalo. I thank all the participants in the courses that I've given, the junior and senior advisors in the programming courses, and the teaching assistants in general. As there are thousands of you, I cannot name and list each of you here.

I am also grateful to all of my co-authors and those contributors whose names do not appear in the articles. Thanks go out to the Computing Education Research groups at the Aalto University and the Tampere University of Technology, as well as to all of my past, present and future colleagues across the globe. Big thanks also to Marina Kurtén ~~that~~ has corrected a huge amount of mistakes in my work.

The financial support from the Technology Industries of Finland Centennial Foundation, the Department of Computer Science at the University of Helsinki, Nokia Foundation, and the Doctoral Programme in Computer Science is also gratefully acknowledged.

Helsinki, October 2017
Arto Hellas

Contents

Chapter 1

Introduction

During the last decades, we have witnessed the digitalization of our society, seen in the large-scale adoption of technologies that keep individuals, businesses, and governments connected through a variety of devices and services. At the core of this change are individuals who understand the nuances that make such devices and services tick, and are able to adjust the behavior of such devices through seemingly arcane methods. These arcane methods are used to conjure opportunities by methodologically chanting and pressing buttons on a slab of plastic. Such conjurers – programmers – are increasingly in demand.

As educational institutions have sought to answer the demand, it has become evident that the task of training someone in the craft of programming is not trivial [77]. This can be observed both in studies that have analyzed programming course retention rates [13, 114, 123], as well as studies on how students perform when working on programming tasks (see e.g. [70,91,101]). When considering the retention rates alone, a survey from 2007 pointed out that, on average, two thirds of students who take on an introductory programming class complete it [13]. To put this into proportion, millions of students take a programming class each year – thousands in Finland alone.

With such statistics, it is not surprising that introductory programming courses receive plenty of attention within the emerging domain of Computing Education Research [95]. Several articles and theses have been devoted to topics such as the use of visualization to help students grasp important concepts [73, 83, 102, 103], tools that are designed to make taking the first programming steps easier [50, 63], systems that help teachers with assessment and facilitate faster feedback to students [53, 54], course design and instruction practices such as constructive alignment for improving the visibility of the learning objectives for both teachers and students [2], practices to assess students' knowledge [108], and pedagogies that help educators to better convey the craft of programming to students [7, 104].

This thesis focuses on ways to improve retention in introductory programming courses and consequently in computing studies, as well as the use of automated assessment tools to support the process. A contemporary interpretation of Cognitive Apprenticeship called *the Extreme Apprenticeship method* and its application is in specific focus, as is the use of a massive open online course (MOOC) to bypass some of the issues related to students struggling with the introductory programming course. In this thesis, "we" refers to both the author and a number of people with whom the author has been lucky to work; "we" varies across different chapters.

1.1 Context and Motivation

This thesis is contextualized within the field of Computing Education Research [76, 95]. It focuses on the theme of retention in introductory programming courses and discusses different ways of teaching programming and how those approaches affect retention.

The case studies have been conducted at the University of Helsinki, Department of Computer Science. The University of Helsinki is a research-oriented university in southern Finland with 36,500 students. As with other Finnish universities, there are no tuition fees and classes are typically organized so that attendance is not mandatory. Students admitted to a university have *a right to study*. Many, however, have families and/or take part-time jobs already during their first-year studies, and cannot always attend instruction.

The majority of the students have attended Finnish primary and secondary schools that, according to a number of studies, provide high-quality education (e.g. PISA [67]). However, traditionally, little to no computing has been offered as a part of primary and secondary school curricula, and even when such courses are offered, they are elective. This means that students have little to no knowledge of what computer science means, and typically have no role models related to the topic. At the same time, students must choose their major as they apply to the university. As such, a number of students enroll to computer science (CS) studies only based on a vague image of what the studies contain [60], and may end up dropping out due to that image proving to be false. These drop-outs are typically most visible in introductory programming courses, where a considerable number of students fail [13, 123].

Even with constructive alignment, guidance, and nationally merited education, at the beginning of this thesis work, the retention rate in the introductory programming courses had been on average 55% over a period of eight years (2002-2009). While this number is lower than the averages reported in international studies (see e.g. [13, 123]), national reports of retention rates from traditional instruction are often on the same level (see e.g. [58, 74]).

1.2 Methodology and Research Questions

Much of the research described in this thesis has followed the principles of design-based research, where the main goal is to positively influence teaching and learning [89]. In design-based research, practitioners start with a research question, and then identify issues – *artifacts* – that are to be improved. The identification of these artifacts is interlinked with the researcher learning about the context under study. After the initial research question and related observations have led to a set of artifacts, the artifacts are worked on in iterations. After each iteration, new artifacts may be identified, which again can lead to new research questions. This natural evolution and refinement is common in design-based research [89, 98].

Our work started with the question *what can we do to help our students learn programming?*, which initially led to a teaching intervention and the creation of the Extreme Apprenticeship method. The Extreme Apprenticeship method is a contemporary interpretation of Cognitive Apprenticeship – a theory of the process of how a master teaches a skill to an apprentice – that attempts to bring the principles of one-on-one instruction to large classes. Cognitive Apprenticeship was chosen as the starting point for the work due to its specific applicability to teaching crafts where the mental processes are typically hidden from view [27, 28], as well as the earlier successes in teaching programming to novices using cognitive apprenticeship [7, 15, 25, 62].

As some of our students were unable to attend local instruction, tools were developed to support learning at a distance. During the process, it became evident that Finnish high schools do not provide sufficient opportunities for programming instruction, which we also sought to remedy. This led to the creation of a MOOC in programming, and the use of the MOOC to recruit students to study Computer Science. From the beginning, the tools that were designed to support learning were also designed to provide data for research. This data has been used, among other things, to illuminate what students do as they are learning to program and what types of automatically identifiable factors contribute, e.g., to the feeling of difficulty during the process has been a constant endeavor.

Overall, the decisions during the iterations have been based on data and observations from the study environment, and the decisions on what to alter and what sort of interventions to apply have been made based on the existing literature and research. Among other results, this work has created evidence-based claims on the applicability of a contemporary interpretation of cognitive apprenticeship to the domain of learning to program, thus answering the call that design-based research requires more than simply showing a particular design works but demands that the researcher generate evidence-based claims about learning that address contemporary theoretical issues and further the theoretical knowledge of the field [9].

The design-based research process has spawned numerous research questions that have been answered during the process. The research questions included in this thesis are as follows:

RQ1: How do existing teaching interventions proposed in the literature influence introductory programming course pass rates?

RQ2: What types of short- and long-term effects in students' course performance does the reorganization of an introductory programming course lead to?

RQ3: How do students admitted through a MOOC in programming perform in their studies when compared to traditionally admitted students?

RQ4: What do students who have never programmed before do when they take their first programming steps and what types of factors contribute to the programming tasks feeling difficult?

Research question one is answered through a literature review, and research questions two and three are answered through case studies organized at the University of Helsinki. For answering research question two, we analyze how the introduction and use of the Extreme Apprenticeship method influenced students' performance in the introductory programming courses as well as subsequent courses. Research question three is answered by comparing the students who have been admitted through the MOOC to students admitted through the traditional entrance exam. Finally, research question four is answered by analyzing source code snapshot data gathered from the students' programming process.

1.3 Outline of the Dissertation

This dissertation is composed of three themes that have all served the purpose of improving the retention rate in the studied context. The first theme in Chapters 2 and 3 discusses the literature on teaching introductory programming with a focus on reported interventions and their influence on course retention. We also discuss a multi-year case study of the application of the Extreme Apprenticeship method, a scalable way to organize introductory programming education with a focus on pragmatic hands-on activities and continuous feedback. For the case study, we focus on the impact on retention rates of the introductory programming courses, as well as on how those students who have taken the course using the Extreme Apprenticeship method fare in their subsequent courses when compared to earlier cohorts.

The second theme, in Chapter 4, discusses extending local instruction to create open online courses available for everyone. We discuss necessary automated programming assessment tools needed to provide feedback to students who due to various reasons cannot attend local instruction. The chapter then continues to describe a solution to the issue specific to the Finnish education system, where students do not always know what they enroll in when they enroll in tertiary level studies. The solution that we studied is a massive open online course (MOOC) that is mainly targeted to secondary level students and is also used as an entrance exam to degree studies. With this study, we have sought to determine if the students' study performance changes when the introductory programming courses have been completed already before entering the University.

The third theme, in Chapter 5, describes some of the research that has been conducted to understand the struggles that students face as they are learning to program. We specifically focus on a narrow set of results that have utilized source code snapshots that have been gathered from the students' learning process, and describe results from the analysis of such data.

Next, as the original publications in this thesis are presented, these thematic areas are denoted with roman alphabets.

1.4 Original Publications and Contribution

This part lists the publications included in this thesis. These articles are organized thematically, where the roman alphabet describes the theme, and the ordinal number describes the order within that theme. Publications

I.1-I.4 include a systematic review of approaches for teaching introductory programming and a multi-year study of the Extreme Apprenticeship method at the University of Helsinki (RQ1 and RQ2). Publications II.1-II.3 discuss extending an introductory programming course as a MOOC, using the MOOC as an entrance exam to university studies, and discusses initial results of students admitted via the MOOC to those admitted via a traditional entrance exam path (RQ3). Publications III.1-2 discuss the very basic approaches and challenges that students face when starting to study programming, and focuses on those students who have explicitly stated that they have never programmed before (RQ4).

I.1 Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success. *In Proceedings of the 10th Annual Conference on International Computing Education Research (ICER '14).* ACM, New York, NY, USA, 19-26.

I.2 Arto Vihavainen, Matti Paksula, and Matti Luukkainen. Extreme Apprenticeship Method in Teaching Programming for Beginners. *In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11).* ACM, New York, NY, USA, pages 93-98.

I.3 Jaakko Kurhila and Arto Vihavainen. Management, Structures and Tools to Scale up Personal Advising in Large Programming Courses. *In Proceedings of the 12th Conference on Information Technology Education (SIGITE '11).* ACM, New York, NY, USA, pages 3-8.

I.4 Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen. Carry-on Effect in Extreme Apprenticeship. *In Proceedings of the 43rd Frontiers in Education Conference (FiE '13)*, pages 1150-1155. IEEE, 2013.

II.1 Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding Students' Learning Using Test My Code. *In Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13).* ACM, New York, NY, USA, 117-122.

II.2 Jaakko Kurhila and Arto Vihavainen. A Purposeful MOOC to Alleviate Insufficient CS Education in Finnish Schools. *Transactions on Computing Education. 15, 2, Article 10 (April 2015)*, ACM, 18 pages.

II.3 Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. MOOC as Semester-long Entrance Exam. *In Proceedings of the 14th ACM SIGITE Conference on Information Technology Education (SIGITE '13).* ACM, New York, NY, USA, pages 177-182.

III.1 Arto Vihavainen, Juha Helminen, and Petri Ihantola. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. *In Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14).* ACM, New York, NY, USA, pages 109-116.

III.2 Petri Ihantola, Juha Sorva, and Arto Vihavainen. Automatically Detectable Indicators of Programming Assignment Difficulty. *In Proceedings of the 15th Conference on Information Technology Education (SIGITE '14).* ACM, New York, NY, USA, pages 33-38.

All the articles have been peer-reviewed and accepted by the computing education research community. The candidate, whose last name changed from Vihavainen to Hellas on 1st of April 2016, has been at least an equal contributor in all of the included articles.

Chapter 2

Teaching Programming at Universities

Introductory programming is taught essentially in all universities that teach Computer Science or related subjects and it is a recurring theme in Computing Education Research. Previously, however, no study had attempted to quantitatively compare the impact that teaching approaches can have on improving the pass rates of programming courses. To remedy this, a review on teaching methodologies was conducted to assess the impact of teaching approaches to introductory programming course outcomes (Publication I.1).

2.1 Retention in Introductory Programming

Table 2.1 contains descriptive statistics of the data (n=32 articles, 60 data points). On average, the pass rates before the intervention were 61.4%, and after intervention 74.4%. The data has plenty of variance, however; the smallest pass rate before intervention was 22.6% and 36% after intervention, while the largest pass rate was 94.2% before intervention, and 92.5% after intervention. The population sizes in the courses also varied considerably. The smallest pre-intervention n was 15 students, which was from a targeted intervention to at-risk students, while the smallest post-intervention n was 9 students; the intervention strategy in the study was applied to a small summer class. The largest number of students was 2298 before intervention, where the study reported data from the past 16 iterations, and 1213 after intervention, which was from a study that reported aggregate results from multiple institutions.

descriptive	min	max	median	mean	sd (σ)
pass rate pre	22.6	94.2	63	61.4	15.5
pass rate post	36	92.5	74	74.4	11.7
students pre	15	2298	148	296.9	487.5
students post	9	1231	86	162.3	200.7

Table 2.1: Pass rates and study sizes before and after teaching intervention.

Five (8.3%) of the extracted data entries had a negative outcome (the pass rates decreased), while in 91.7% of the entries the intervention had at least a minor improvement on the overall results. On average, the interventions improved the pass rates 33.3% or nearly one third.

2.2 Interventions and Improvements in Retention

The articles in the final data (n=32 with 60 data points) were analyzed, and the teaching interventions were manually coded to identify the components in each intervention (see publication I.1 for further details on the coding process). The ten most frequent intervention codes encompassed the following activities:

- *collaboration*: activities that encourage student collaboration either in classrooms or labs
- *content change*: at least parts of the teaching material was changed or updated
- *contextualization*: activities where course content and activities were aligned towards a specific context such as games or media
- *CS0*: the creation of a preliminary course that was to be taken before the introductory programming course; could be organized only for e.g. at-risk students
- *game-theme*: a game-themed component was introduced to the course, e.g. a game-themed project
- *grading schema*: a change in the grading schema; the most common change was to increase the amount of points rewarded from programming activities, while reducing the weight of the course exam
- *group work*: activities with increased group work commitment such as team-based learning and cooperative learning

intervention code	n	min	max	median	avg	σ
collaboration	20	-1	59	39	34	17
content change	36	-17	69	34	34	17
contextualization	17	18	69	37	40	17
CS0	7	18	76	41	43	19
game-theme	9	-39	42	21	18	23
grading schema	11	3	42	30	29	12
group work	7	36	59	44	45	7
media computation	10	24	69	49	48	16
peer support	23	-1	59	36	34	16
support	9	-29	67	36	33	19

Table 2.2: Ten most common intervention codes and the overall intervention effects of the studies in which they appeared in. Number of studies including the intervention denoted as n, realized pass rates reported using minimum, maximum, median, average and standard deviation (σ) in percentages.

- *media computation*: activities explicitly declaring the use of media computation (e.g. the book)
- *peer support*: support by peers in form of pairs, groups, hired peer mentors or tutors
- *support*: an umbrella term for all support activities, e.g. increased teacher hours, additional support channels etc.

Table 2.2 contains the ten most frequent codes and the realized gains in the studies in which they appeared in. While the intervention types cannot be compared with each others due to overlapping, the table provides an overview of the realized improvements over different studies.

When considering the median improvement, the studies that had media computation as one of the components were most successful, while studies with a game-theme were the least successful. Facilitating group work and collaboration, and creating a preliminary programming courses that was offered before the actual introductory programming course were also among the high-performing activities. While the effect of an intervention activity depends naturally on other activities as well, a noticeable amount of variance was observed even within similar setups. This variance is caused, at least, by the natural variance of student populations over different semesters, student intake, teacher effect, difference in grading criteria among different institutions, and the difference in student workloads between different institutions.

The teaching interventions were further categorized into five categories; (1) collaboration and peer support, (2) bootstrapping, (3) relatable content and contextualization, (4) course setup, assessment and resourcing and (5) hybrid approaches.

Collaboration and Peer Support

Approaches that include collaboration and peer support include peer-led team learning activities [65], pair programming activities [126] and cooperative and collaborative practices [26, 122]. A total of 14 studies were classified as having applied an intervention, which primarily consisted of moving towards a collaborative, or peer support based approach. Three specific approaches were identified: cooperative learning (3 courses), team-based learning (5 courses), and pair programming (6 courses).

Cooperative learning was found to yield the largest absolute improvement in CS1 pass rates (25.7% on average), and team-based learning was found to yield the second largest absolute improvement (18.1% on average). Despite being frequently cited as an enabler for programming skills, the pair programming approach was only found to yield an absolute improvement of 9.6% on average, and ranked 11 out of the 13 interventions that were explored by this study. It was possible that courses to which this intervention was applied already had good pass rates, and therefore there was little scope for absolute improvement.

When considering realized changes, pair programming yielded a realized increase of 27% in pass rates on average, but overall, this approach was still ranked 11th out of the 13 interventions which were explored by this study. Considering the results of all 14 courses combined, we found that instructors who applied a collaborative or peer support based intervention generally received the largest improvements in pass rates when compared to the other groups examined in this study (16.1% improvement, realized change 34.3%). A possible explanation is that collaboration increases commitment as the student is no longer responsible solely to herself, but also to her peers. Collaboration can also increase feedback opportunities, and create situations where the student may learn from helping others. At the same time, expected collaborative activities may also lead to a situation, where some students who can not attend local instruction due to e.g. other commitments are less likely to attend the course, if it is not mandatory for them.

Bootstrapping

Bootstrapping practices either organized a course before the start of the introductory programming course [45, 97] or started the introductory programming segment using a visual programming environment such as Scratch or Alice [72]. Some of the activities were also targeted at at-risk students [85]. A total of 9 studies were classified as having applied such an intervention. Two specific approaches were identified: using visual programming tools such as Scratch or Alice (5 courses), and introducing CS0 (4 courses). Out of all the interventions that were explored in this study, using visual programming tools was found to yield the fifth largest absolute improvement in pass rates (17.3% on average). A similar high ranking was found when considering realized improvement (fourth, 38.6%), which positioned using visual programming tools as the fourth overall best intervention.

Whilst the absolute improvement for courses that introduced CS0 was much lower than visual programming (10.5% increase), the realized change that was yielded by this intervention was comparable (34.9% increase). Considering the results of all 9 courses combined, we found that instructors who applied a bootstrapping intervention generally received the second largest improvements in pass rates when compared to the other groups examined in this study (absolute change 14.3%, realized change 37.0%). It is possible that the initial simplification offered by these forms of intervention are able to assist students who might otherwise fail CS1, by suppressing the syntax barrier until they have gained sufficient knowledge of the underlying concepts. This also ties into research on threshold concepts, which suggested that reducing the level of complexity initially may be an effective way to assist students in overcoming thresholds.

Relatable Content and Contextualization

Approaches that introduced relateable content sought to make programming more understandable to students. These approaches include media computation [109], introducing real world projects [30] as well as courses that evolve around games [10]. A total of 14 studies were classified as having applied an intervention, which primarily consisted of using relatable content and contextualization as a means to improve CS1 pass rates. Two specific approaches were identified: media computation (7 courses), and gamification (7 courses).

Out of all the interventions that were explored in this study, using media computation was found to yield the seventh largest absolute improvement in pass rates (14.7% on average), and a comparable improvement was found for gamification (10.8% on average). However, when considering realized changes, media computation was found to yield the largest realized change across all interventions explored in this study (50.1% increase), whereas gamification was found to only yield the tenth largest (27.4% increase). Overall, and considering the results of all 14 courses combined, we found that instructors who applied a relateable content or contextualization intervention generally received the third largest improvements in pass rates when compared to the other groups examined in this study (absolute change 11.6%, realized change 38.7%). As media computation (overall rank 2) considerably outperformed gamification (overall rank 10), it could be the case that whilst games provide a useful tool to contextualize a learning task, there are still fundamental underlying programming concepts that can be better served by adopting a media computation approach.

Course Setup, Assessment, Resourcing

Approaches that modify course setup, assessment and resourcing included a broad range of practises starting from adjusting course content based on data from an assessment system [92], introducing new content, a programming tool that provides additional support and changing the grading schema assessment [74, 82]. A total of 15 studies were classified as having applied an intervention which primarily consisted of changing aspects of the course setup, rather than changing elements of the teaching approach. Three specific approaches were identified: changing class size (4 courses), improving existing resources (2 courses), and changing assessment criteria (9 courses). Overall, the largest absolute improvements in pass rates were found by changing the class size (17.8% improvement) and improving existing resources (17.5%).

However, when considering these improvements relatively, they were among the five worst interventions found by this study. Similarly, making changes to the assessment criteria applied in the course yielded on average an absolute improvement of 10.1% and realized improvement of 22.5%. But when considering these changes against the other 13 interventions explored by this study, changing assessment criteria ranked 12th. Considering the results from all 15 courses combined, we found that instructors who applied an intervention based on course setup generally yielded the fourth largest improvements in pass rates when compared to the other groups (absolute change 13.4%, realized change 26.8%). The findings on changing class size to improve pass rates are consistent with previous studies [13,123] that have suggested that smaller classes generally have lower failure rates than larger ones. However, overall, it is possible that this group of interventions was ranked as one of the lowest because making changes to the course setup, such as the assessment criteria, does little to adjust the likelihood of a student overcoming thresholds understanding programming concepts.

Hybrid Approaches

Hybrid approaches are approaches that were not included in any of the primary categories. These include combinations of different practices [64, 81, 96]. A total of 8 studies were classified as having applied an intervention, which primarily consisted of combining several different teaching interventions to yield a hybrid approach. Three combinations were identified: media computation with pair programming (2 studies), extreme apprenticeship (3 courses), and collaborative learning with relateable content (e.g. games) (3 courses).

Overall, combining media computation with pair programming, or adopting an extreme apprenticeship approach were found to yield mid-range improvements in pass rates, ranging from 13.5-16.5% in absolute terms, or 36.9-49.3% in realized terms. These approaches were ranked fifth and seventh among the overall 13 interventions that were explored in this study. However, combining collaborative learning with relateable content was found to be the worst overall intervention, actually yielding a decrease in pass rates of 9.7%, or 53.7% in realized terms. However, we note that some of the courses, which switched to this approach already had a very high pass rate ($> 90\%$), and therefore the scope for improvement was minimal.

2.3 Notes on Comparing Different Interventions

The final question in the review was to determine whether there were any significant differences in the post-pass rates of studies that applied different types of interventions. Grouping the 60 post-intervention pass rates by the five primary intervention categories, a statistical analysis suggests that whilst substantial improvements in pass rates can be achieved by applying different interventions, the overall pass rates after applying different types of intervention are not substantially different.

Overall, it is challenging to analyze the effect of a pedagogy or teaching improvement. When looking from the administrative point of view, one often examines the effect broadly by e.g. investigating student throughput and number of degrees, cost-effectiveness [14], faculty readiness to adopt new teaching methods (see e.g. [40,100]), while not being able to extract the true effect of the improvements in the learning within the inspected course.

On the other hand, when looking from the relatively narrow point of view of a teacher, the reports often emphasize the uniqueness of the course without inspecting the effect on the subsequent courses [11,65]. However, extending the view from a single course is important as the expertise accumulates throughout the degree.

During the study, we found almost no reports on interventions that did not yield an improvement. It is possible that educators who have tried an intervention but received poor results simply have not been able to publish their results due to a negative result. Similarly, the teaching interventions almost always focused on a single course, and very few studies included data from subsequent courses. It would be meaningful to study results from long-term application of a teaching methodology, as well as consider the implications of such interventions to students in their subsequent studies.

Chapter 3

Extreme Apprenticeship method

In this chapter, we focus on a particular teaching approach called the Extreme Apprenticeship method that has been used at the University of Helsinki. We discuss the method, and how its application has influenced the retention in the overall studies (Publications I.2-4).

The Extreme Apprenticeship method stems from Cognitive Apprenticeship, a theory of instructional practices used for teaching cognitive skills and working practices that are invisible to the learner [27,28], and Extreme Programming, a software development discipline that values software quality, responsiveness to changing environments and customer demands, and working only on the requirements that are truly needed [12]. Extreme Apprenticeship focuses on building the industry-relevant skills and professionalism of students from day one, scales the amount of individual instruction to the extreme, helps to facilitate one-to-one instruction in classes with hundreds of students, and, at the same time, reduces the overall costs related to facilitating education. From the management perspective, one seeks to reduce the waste of resources – including the always-limited time of students and instructors – whilst at the same time focusing the available resources on activities that are known to help learn the topic at hand.

Adopting practices from Cognitive Apprenticeship to introductory programming education is not a novel idea. On the contrary, it has been used in teaching programming with promising results for nearly two decades [7, 15, 25, 62]. Cognitive Apprenticeship takes the model of apprenticeship education, where a senior practitioner – a master – employs novices as a workforce to help in on her daily tasks. At first, the tasks are simple, but as the novice learns to handle the simpler tasks, more challenging work is given. While the apprenticeship model has mostly been applied to manual professions where the work is visible to both the learner and the instructor (e.g. carpentry, c.f. [47]), the Cognitive Apprenticeship theory emphasizes learning cognitive skills and processes that are invisible. Many of the practices are about making the thinking processes of a more experienced practitioner visible to the novices – and vice versa.

Cognitive Apprenticeship theory discusses a number of steps that are important in this interplay of thoughts; modeling, coaching, scaffolding, articulation, reflection, and exploration. In the *modeling* stage the teacher provides students with a conceptual model of the process e.g. by showing how something is done or by providing guidelines that the students can follow. When the students have acquired a *mental model*, they start working under the guidance of the master. The master *coaches* students as they work on their assigned task, and provides *scaffolding* when needed. Scaffolding – or instructional scaffolding – refers to facilitating and supporting a learning process that promotes a deeper level of learning, which is tailored to help the student reach her goals [22, 88]. The idea of scaffolding is related to Vygotsky's zone of proximal development [121], which is an area of knowledge or skill that the learner cannot reach by herself, but can achieve with the support from others. As learners can learn and adapt to new situations and knowledge, the zone of proximal development is learner- and time-specific, and thus, the scaffolding must be constantly adapted.

A part of the learning and teaching process is making the students' thoughts visible, and helping them to understand the ramifications of the choices. This is facilitated by having students *articulate* their solutions or thoughts as they are working. This process can be made more explicit by e.g. asking the students questions, interspersing scaffolding into the articulation process. Questions that encourage students to create meaning to their task via e.g. self-explanations [118] can be both used to guide students' thinking towards a desired outcome, as well as to help students focus their thoughts. If students are shown the works of others, or provided with sufficient hints or guidance, they can also *reflect* on their thought process and solutions, and possibly, compare it to that of others.

Once the students have mastered smaller steps that they are expected to complete, they are given the opportunity to *explore*. This room for exploration can be given by e.g. providing open-ended assignments, as well as adapting scaffolding. Although there is no limit on the amount of guidance that a student can receive, or on the number of times that the assignments can be returned, it is of utmost importance that as soon as the student does not require scaffolding and can proceed on her own, the scaffolding is faded, i.e. the support is reduced so that the student does not become dependent on it. A cycle that involves many of these steps takes place several times each week as each week typically contains several learning objectives and tens of assignments.

When comparing to existing implementations in the domain of teaching programming, Extreme Apprenticeship is mainly targeted for the domain of introductory programming, takes the effort of an individual student to the extreme, minimizes instructor-driven learning, and focuses also on the organizational aspect of facilitating instruction. The main learning method is *doing*, and all instruction evolves around maximizing the amount of meaningful effort that a student puts into working on assignments, as well as supporting the *learning of meaningful working practices*. Instead of a traditional lecture-driven model, where the student attends a lecture – and perhaps answers a few quizzes together with other students – lectures provide only the bare minimum information needed to start working on the assignments. As learning starts on the very first day of the course, students start working on assignments on the same day. Dozens of programming assignments that offer a stable progress are provided to the students from the start. The students work on the assignments in a suitable space, where the instructors coach and scaffold students.

3.1 Values and Practices

To help instructors adapt the Extreme Apprenticeship method into their own context, a loose set of values and practices are provided in Publication (I.2). These values and practices are rephrased below for the reader.

Values

When planning and executing instruction, *all activities should be aimed towards helping students become professionals in the craft they are learning*, and executed so that the *students' working processes are guided towards those used by professionals*. Motivation is crucial; one should help build up the inner fire of a learner that feeds continuous improvement. This supports and nurtures the growth of a mindset that embraces change and deliberate, personal effort that needs to be taken to learn. The path towards becoming a professional starts from day one, and the instructors need to intermittently align their practices, tools and methodologies with this explicit goal in mind.

The organization that facilitates instruction – from teachers to administration – needs to *implement continuous feedback practices where the information flows into both directions*. The student receives feedback from the learning experiences, instructors and peers as she progresses, and the instructor, material and tools guide her to improve her own practices. The instructor monitors the student's progress and challenges, and – if needed – organizes, creates or facilitates scaffolding by e.g. altering the student's learning path, materials, or by providing just enough hints so that the student is nudged in a direction where she can proceed. Moreover, data is gathered to analyze long-term effects of choices made during the learning process, and to make long-term analyses and organizational decisions, based on which instruction can again be adapted to better fit the context.

This feedback is also used to observe students' progress as they are working. If one notices that a student has not acquired enough experience and practice in a specific topic, additional work can be provided. Any *skill is only mastered by actually practicing it*, and the instructor must provide plenty of opportunities for such practice. These opportunities must be adapted to the learner's level so that there are both opportunities for practice as well as challenges where students need to search additional information in order to proceed.

Every student is expected to live up to a set standard of the organization, and *no compromises* are to be made. These standards can be set by using learning objective matrices that outline expected competences and learning outcomes, assignments that need to be completed, as well as traditional assessment methodologies where students can learn and measure their skills. If a student has not reached the expected learning outcomes and needs more practice, having the student proceed to challenges that she is not ready to face is likely to lead to a failure. Instead of making the progress of a student who is struggling easier, an instructor should seek to make the challenges explicitly visible while at the same time supporting the student to learn from the mistakes and challenges – akin to Fail Fast [94] – so that the foundational knowledge that is needed to proceed is kept solid.

Practices

When providing instruction, an instructor should *avoid preaching* – that is lecturing by e.g. reading out loud the instructional materials – and only provide the minimum amount of information for the learners to start with the assignments. If most of the instructor's time is spent on lectures, preparing for lectures, and on answering e.g. students' emails, it is often poorly used. Most of the instructor's time should be directed to one-on-one interaction with students who are working on course assignments, and to improving learning materials and assignments based on the observations and feedback. Lectures should *cover only the minimum of what is needed to get started with weekly assignments*, and the *examples should be relevant to the assignments.*

Work in a course – as well as in tertiary education – should *start early.* Assignments and learning materials are to be provided to students in a timely fashion – both encouraging eager students to start on the next set as well as to avoid burdening students who schedule their work closer to the deadline. Students should be directed to work on the assignments as soon as possible, and they should have completed a considerable number of assignments already during the first days of a course. This early and visible success helps in building both a strong routine, as well as increases students' self-esteem [18]. As much of professionalism and skill is about the countless number of hours put into training and deliberate practice [36, 75], opportunities to *train the routine* are needed; the number of assignments should be high and, to some extent, repetitive to increase students' programming routine.

When the students start to learn a new topic, the assignments and material must *provide clear starting points and structures on how to start solving the task, as well as clear guidelines on how to proceed.* Assignments that are split into smaller parts with clearly set intermediate goals support students as they are learning new constructs and practices; these small intermediate steps help students feel that they are learning and making progress when working on otherwise new concepts and topics. The assignments and materials are a part of provided scaffolding, and when designing assignments, the goals should be made explicit.

At the same time, when designing the assignments, the scaffolding needs to also be dismantled based on students' expected knowledge – which can vary – thus, open assignments are also mandatory. Open assignments typically do not guide the students in designing the structure and provide freedom in design choices. Instructors naturally help and provide feedback on students' choices.

Overall, the interaction with materials, assignments and peers is only a part of the instruction, while the other part is the interaction with course staff and other individuals. The organization needs to provide the facilities that make it possible to have *help available* – a big part is to have a lab where students can work on the assignments in the presence of instructors and peers. Furthermore, the expectations should be made explicit, and as the assignments are the main instrument in learning, the majority of the *assignments need to be mandatory* to all students – there are no shortcuts to learning. As the students learn, the level of guidance from the materials and the peers can gradually decline in order to *encourage to look for information.* Whilst at first, students are helped in the core tasks, they gradually also are given more room and freedom to explore, which in turn cultivates the information-seeking process that is a crucial skill of any professional. In essence, students need to be expected to find out things that are not covered during the lectures or in the material.

By following these values and practices, the instructor can create student-centric assignment-driven instruction with high amount of one-to-one interaction. When adapting the Extreme Apprenticeship method to a specific context, these values and practices are to be followed until the instructor is able to inspect the impact of the method on the students. After this, context-specific adaptation based on the inspected outcomes can be made.

3.2 Scalability

Providing one-to-one support and instruction is inherently resource-dependent as practice needs a space with appropriate tools and feedback, while support requires experienced personnel. Having sufficient space and hardware is context dependent. At the University of Helsinki, the space has not been an issue – most of the computer labs have been heavily underused in the past, and many of them have already been dismantled due to the lack of use. Most of the students have a laptop, and to further help in the hardware issue, our department provides new students with laptops at the very beginning of their studies. The computer labs also have sufficient hardware. Software, when chosen in a sensible fashion, is also not an issue – workstations can be equipped with open-source platform and tools, e.g. a Linux distribution. Furthermore, reducing the amount of lectures reduces the overall facility costs, as well as frees up the time of the lecturer.

Providing continuous feedback is more challenging to scale up without a show-stopping increase in costs. As the practices in the Extreme Apprenticeship method require a larger amount of feedback than traditional instruction practices, it is important that the use of resources is optimized over time dynamically by using tools, structures, and even voluntary human resources. Automatic assessment of programming assignments provides a somewhat limited solution due to the lack of focus in the *working process*; one specifically tailored tool is discussed in Chapter 4.

One solution to the scalability issue is to use voluntary student advisors to help in providing feedback. The students who participate as advisors become legitimate peripheral participants [66] of the teaching community, which is beneficial for all parties. They may increase retention [107] and improve the learning context atmosphere [31]. At the same time, as the students gain experience from teaching, the faculty gains knowledge of each student's strengths, based on which they can be e.g. recruited as teaching assistants or research assistants.

At the University of Helsinki, we have sought to have some 10-20% of our students as advisors. Such activity requires a coordinator, whose task is to manage day-to-day activities and to allocate sufficient resources to appropriate situations. The coordinator also takes a role in the recruitment of advisors for the upcoming courses. While in the first course iterations, the coordinator was a faculty member and received no compensation for the task in our context, in the latter course iterations the role has been given to a senior student who has been advising for a number of times.

Students can participate in three different roles. During the first semester of advising, the student acts as a *junior advisor*, who helps others in the lab and learns about teaching and learning. We chose to compensate the junior advisors in the form of study credits, as learning to coach is essentially a study experience. If a student opts in for another semester of advising, she may be recruited as a more experienced *senior advisor*, who also receives monetary compensation. After participating as a senior advisor, one of the senior advisors is selected for a third semester to act as the advisor coordinator – which also is a role that is monetarily compensated. Many students are recruited into other junior (non-tenure) faculty teaching positions after they have worked as senior advisors.

As having a large portion of first-year and second-year students participate as advisors became a more natural activity at the department, guidelines for teaching were established. A lab manifesto was published to the advisors, and updated as experience of proper advising principles grew during the courses. The manifesto states a few guidelines as pedagogical practices and takes a stand on both the instruction practices as well as guiding the advisor on how to behave in the class. In addition, advisors are given guidelines on resource usage.

While the intensive one-on-one instruction was expected to be more burdensome for the instructors, as has been reported related to mastery learning (see e.g. [42]), a survey conducted by us indicated otherwise. We anonymously surveyed advisors who have worked both as traditional teaching assistants as well as advisors. The survey indicated that advising in labs was seen as much more rewarding and meaningful, and that the advisors considered the students' rapid and visible progress as efficient use of advisors' time. Many pointed out that the experience was so rewarding, that they volunteered (or "chilled out") in the lab and advised students also just for fun (for additional details, see Publication I.3 and [119]). Note that while the overall cost – even with a significant increase in the number of assistants – is comparable to the traditional teaching model due to facility management and course organization (Publication I.3).

With the influx of students who first participate as voluntary advisors and learn to advise, after which they may apply to other positions, the responsible faculty member only needs to be a part of the teaching team – not the force of all instruction. When good enough teaching materials and practices are in place, and there is sufficient support for the advisors, it is completely acceptable that the advisors are not as experienced as traditional teaching assistants are.

3.3 Success in Subsequent Courses

The change from the traditional teaching method (lecture-based with take-home assignments) resulted in a statistically significant change in acceptance rates of our programming courses; the average rates of our introductory programming course and advanced programming course have increased by 32% and 37% respectively (for additional details, see Publication I.4 and [115]) over a period of three years. As discussed in Chapter 1, this improvement was made over a situation where a lot of effort had already been invested into the improvement of the introductory programming courses.

In the article "Carry-on Effect of Extreme Apprenticeship" (Publication I.4), we investigated the carry-on effect using three different measures: (1) credit accumulation 7 and 13 months after the start of students' studies, (2) success in the expected study path during the first semester by examining the success in two of the subsequent courses right after the first programming course, and (3) grade distribution in the first mandatory programming course. The comparison was made between the populations that started their studies with Computer Science as major subject during 2007-2009 and 2010-2012. Between 2007 and 2011, the instructor in charge of the introductory programming course was the same, but in year 2012, the instructor was changed. The teaching approach was changed from a traditional approach to the Extreme Apprenticeship method in 2010. The 2012 cohort includes students who attained study positions through a MOOC in programming (discussed in more detail in Chapter 4).

Table 3.1 includes details on students' credit gains during the interval of 7 and 13 months, based on the year when they enrolled at the university and started their studies. Only students who attempted at least one course are included. The table includes number of students and sum of credits after 7 and 13 months of studying for each group. In addition to the sum, normalized credit counts are also shown. The normalization is calculated based on the student population, and year 2008 is used as the baseline as the student intake was decreased from 2007 by 20 students. The table is depicted as it was shown in the original article.

Year	Students	Credits 7 (norm, scaled %)	Credits 13 (norm, scaled %)
2007	136	1681 (2237, **91.6**)	2558 (3404, **95.2**)
2008	119	1605 (2441, **100**)	2352 (3577, **100**)
2009	120	1616 (2437, **99.9**)	2686 (4051, **113.3**)
2010	136	2030 (2701, **110.7**)	3418 (4549, **127.2**)
2011	140	2287 (2957, **121.1**)	3352 (4334, **121.1**)
2012	168	3042 (3277, **134.3**)	–

Table 3.1: Credit accumulation of student groups [59].

Table 3.2 displays the percentage of students who successfully transition from the first mandatory course (Introduction to Programming) to the two subsequent mandatory courses (Advanced Programming and Software Modeling). All of these three courses are mandatory for every starting CS student, and all of them are scheduled to be taken during the first semester of studies. Before the change to the way introductory programming is currently organized, less than half of the students were successful in completing their first mandatory courses on their first attempt, and after the change, almost 70% of the students completed the courses successfully.

Year	Intro Prog. → Adv. Prog	Intro Prog. → Software Modeling
2007	45.1	41.5
2008	39.2	48.8
2009	50	54.2
2010	68.5	63
2011	71.1	74.4
2012	70.3	72.2

Table 3.2: Percentage of students that successfully complete mandatory courses on their first attempt, described as course pairs [59].

Overall, a statistically significant improvement over the baseline was observed both in the introductory programming course, as well as in the student credit gains. This is something that can not only be explained through improved retention in the introductory programming courses – the success in subsequent courses suggests that students have a better grasp of the required skills than previously. For additional details, as well as information on the change in grade distribution, consult Publication I.4.

Chapter 4

A MOOC Focused on Programming

Here, we discuss an automated assessment suite called Test My Code (TMC) that is used to facilitate some of the one-on-one scaffolding in programming courses (Publication II.1). Then, we outline the MOOC activity in more detail and discuss how pupils in secondary education are recruited to the University of Helsinki through the MOOC (Publication II.2). Finally, we discuss the results of the first study that compared students who were admitted through the MOOC to the CS programme to students who were admitted to the CS programme via the more traditional entrance exam (Publication II.3).

4.1 Scaffolding Students' Work from Distance

In order to properly benefit from an automatic assessment system in distance education, the system has to accommodate scaffolding as well as support the situative perspective on learning in apprenticeship-based education [19]. The perspective views the situations where knowledge is developed and later applied as highly connected, as "methods of instruction are not only instruments for acquiring skills; they also are practices in which students learn to participate" [41]. According to the situative perspective, abstracting theory from practice does not yield transferability [29]. As a goal of instruction is to help students on their journey to becoming experts in their field, the chosen tools and methods have to allow "participation in valued social practices" [41] of the respective professional communities.

For aspiring programmers, the tools and *practices of learning* in their training should be similar to those used in the software engineering industry.

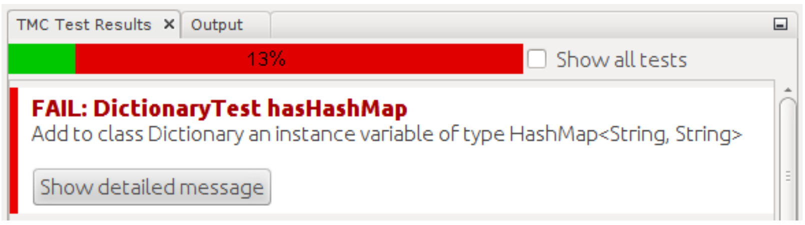
Even in distance education, the main activity in learning to program should be programming, using instruments that are relevant to the practice such as integrated development environments.

While a vast number of systems for automatically assessing students' programs exist [1, 33, 54], some of them created specifically for providing faster feedback to the students [105], the emphasis on retaining the workflow of a professional has not received enough attention. Traditional automated assessment systems suggest a flow, where the student needs to download a compressed template from a web-page, extract it, open it within the IDE, solve the task, compress the solution files, and then submit the compressed solution using a web-interface.

At the same time, as scaffolding should be well-timed and not overly excessive, having students go through an external system to receive feedback on the progress could be improved. Ideally, the students should receive feedback already within the environment that they program in.

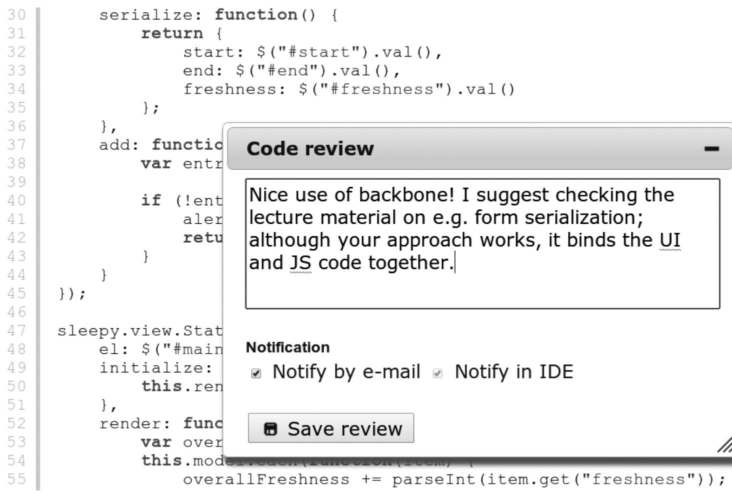
The contribution in Publication II.1, Test My Code (TMC), seeks to solve some of these issues. In addition to providing traditional automatic assessment capabilities, book-keeping facilities and grading support, TMC can provide support to students directly within the environment that they are working in. The students' tasks are retrieved to the programming environment – here, we have explicitly chosen to build upon industry standard IDEs such as NetBeans – and the work can be submitted directly from the IDE. This enables a workflow that is similar to many modern workflows, where a software developer first pulls her tasks from a version control system, performs the tasks that she needs to do, and then commits her work back to the version control system. Once the work has been committed, an integration server runs a number of tests on the changes to provide feedback for the developer. As TMC is integrated to an IDE, local testing facilities are also in use, which makes it possible to provide rapid step-by-step support and feedback for the student.

Figure 4.1 shows a scaffolding message given to a student as she needs to create a dictionary that utilizes a HashMap. As her solution is missing the expected HashMap, TMC tells the student to create one. This scaffolding is provided directly within the working environment with no need for external systems or a visit to an external web page. The assignments are constructed by an instructor or a course designer, and the feedback and scaffolding is typically exercise-specific. TMC provides an API that makes it easier to create scaffolding messages for exercises. In addition to the local guidance, TMC provides capabilities for distance guidance via e.g. code reviews [38]. The code reviews are typically made within a web page – see Figure 4.2



FAIL: DictionaryTest hasHashMap
Add to class Dictionary an instance variable of type HashMap<String, String>

Figure 4.1: *Test My Code provides students with step-wise guidance within the programming environment helping students design their work appropriately. This feedback is constructed using an API that is provided to the instructor, and can be used to e.g. guide class design or to do more traditional input-output testing.*



```
    serialize: function() {
        return {
            start: $("#start").val(),
            end: $("#end").val(),
            freshness: $("#freshness").val()
        };
    },
    add: functio
        var entr

        if (!ent
            aler
            retu
        }
    }
});

sleepy.view.Stat
    el: $("#main
    initialize:
        this.ren
    },
    render: func
        var over
        this.mode
            overallFreshness += parseInt(item.get("freshness"));
```

Code review

Nice use of backbone! I suggest checking the lecture material on e.g. form serialization; although your approach works, it binds the UI and JS code together.

Notification
☑ Notify by e-mail ☑ Notify in IDE

Figure 4.2: *The web interface of Test My Code provides the ability to perform code reviews on students' solutions.*

for an example. The reviews that are created by instructors, course staff or students are then made available for the student via a notification in the programming environment or in an e-mail – note that the use of such functionality is meaningful in our context as there are plenty of junior advisors who learn also from conducting code reviews.

In addition to the guidance facilities, TMC can collect key-level data from the students' learning process, which can be used to study how students construct solutions to programming problems (Chapter 5 discusses this topic). We have also published tools that can be used to study the students' learning progress. For further information, see e.g. CodeBrowser [48].

While the move to the use of an automated assessment system that provides scaffolding as students are working has been rather mandatory for successful facilitation of online education, it has its downsides as well. Our experience has shown that increasing the use of automated assessment has reduced both learner-to-learner interaction that helps learning programming and learning interpersonal skills, as well as learner-to-teacher interaction, which has been beneficial for students as they have feedback opportunities and for teachers as they can see and fix the issues that students face. Both are known to be important in online settings [6]. While this may be acceptable to some level in order to reach specific goals in distance education, one needs to take care to ensure the meaningfulness of local instruction, both for the students and the student advisors.

4.2 Open Online Course in Programming

The Finnish education system provides schools with some freedom in what elective courses to offer and how to organize teaching in classrooms. This means that schools with eager teachers can certify third-party computing courses if they choose to do so.

Since, at the time of writing this thesis, the curriculum in Finnish schools does not include computer science or related studies in general education, pupils who are interested in learning computing-related topics rarely have the opportunity to study such subjects at their local schools. This lack of computing-related topics means that many pupils fail to understand the importance and dominance of computing in their everyday lives, and at the same time may not be giving the topic the value it deserves. They also do not know what computer science is, where it can be studied, and whether studying it would suit them or not.

To remedy this issue, and to bypass bringing CS-related topics to the national or regional school curricula (see e.g. [80, 111] for challenges), we have offered our programming courses for all pupils in Finnish schools. Schools that choose to offer our courses for their pupils do not need to have a proficient teacher, only a teacher or a staff member who acts as a supervisor in a possible course exam. To encourage teachers to take our offer, we provide the course exam as well as mark it free of charge. At the same time, additional incentives for completing the course fully are offered; if a pupil completes a required number of weekly programming tasks, she is invited to an interview, which can lead to a full study right in CS at the University of Helsinki.

This arrangement has been in operation since January 2012, and a course with the option of securing a right for degree studies has since been offered each spring. Six student populations (2012-2017) have started their studies via the online course. In addition to the student recruitment, we have stimulated the national discussion on the opportunities and benefits of including CS-related topics and areas into the Finnish school curriculum, as well as created discussion on alternative admission approaches.

As the course is open for all, no additional material than that what is provided on the course web-site is required. Programming assignments and a blended online material form the core of the course, and the learning objectives are embedded into the assignments. All work is done with authentic tools with the support from Test My Code as well as the learning materials. As there is no need to provide other information than a valid email address and to download the programming environment, participants have indicated that starting the course is easy.

4.3 MOOC as an Entry Mechanism to Degree Studies

In the first online course that we offered in early 2012, the participants had to do at least 80% of the weekly programming assignments in order to be able to attend an interview at the University of Helsinki and to apply for a study right. The course had up to 30 weekly programming assignments. The interview contained both a programming assignment as well as a discussion with two members of the Computer Science department. A majority of the participants who came to the interview performed well, and were able to complete the given programming assignment. Only a handful of participants had issues with e.g. program design or did not reach a working program at all. Out of 52 participants who applied for a study right during Spring 2012, 49 study rights were granted. From the 49 participants, 38 started their studies during the following semester. The remaining had varying reasons not to start their studies: they were still continuing their studies in secondary education, they postponed the start due to the mandatory military service, or they took another study right.

When taking an administrative view to the students' course records, even while admitting "almost everyone" who worked through the introductory programming course, the admitted students are at least on par with the normal intake. Table 4.1 shows an administrative comparison of the study success of those students who started via the open online course ($N = 38$) to those who started through a more traditional path ($= 68$),

	CS Courses		**All Courses**	
	MOOC	NORM	MOOC	NORM
Credits				
mean	33.08	23.96	44.08	33.76
standard deviation	11.32	15.3	17.58	21.96
Courses passed				
mean	9.11	6.65	11.42	8.81
standard deviation	2.95	4.02	4.24	5.27
Courses failed				
mean	1.74	2.31	2.66	3.15
standard deviation	1.83	2.37	2.16	2.73
Grade (scale 1-5)				
mean	4.04	3.79	3.94	3.73
standard deviation	1.15	1.2	1.13	1.18

Table 4.1: Comparison of student performance during the first year of studies. Data includes students from the first MOOC ($N = 38$) and the students who started their studies through the traditional admission path ($N = 68$). Study records between 1st of August 2012 and 24th of May 2013 are included (Publication II.3).

i.e. entrance exam, matriculation examination score, or a combination of both. The comparison excludes the students who received the study right but did not attend any courses, and the students who gained the study right through the traditional path and took the introductory programming course as a MOOC during the summer before the start of their studies. In the comparison, the students who were admitted via the online programming course fare marginally better during their first year, credit-, course- and grade-wise.

Since 2012, due to increased awareness and interest, gaining the study right via the open online course has become more challenging. Whilst practically everyone who applied through the open course in 2012 was given a study right, the number of participants has since annually increased and the proportion of accepted participants consequently decreased. At this point, we are still in a situation where the long-term benefits of admitting students via an open online course in programming are yet to be studied.

We acknowledge that admission is just a single step, and studying the admission system as an entity is challenging. The entrance criteria may change over the years due to changing participant counts. Similarly, students admitted through the MOOC no longer face the “traditional” hurdle in computer science studies – that is, the introductory programming course – but, there are surely other challenges. These challenges, however, manifest themselves over the years, and include things such as getting employed.

When building such a path as ours, the educational system needs to support the transition from secondary education to tertiary education for more students to succeed in degree studies [125], regardless of the admission path. In our context the path for students who start their introductory programming courses during the first semester has been well planned and laid out, but organizing the courses that the MOOC intake takes in a sensible fashion still requires work. Currently, only little effort has been put into the MOOC cohort, i.e. offering a guided self-study CS2 course during the first semester.

Chapter 5

Mining Student Data

In the same way as Extreme Programming focuses on delivering additional business value in every software iteration, an educator should focus on delivering additional educational value and improvements in every course and teaching iteration. When proper feedback processes are in place and evidence is gathered from every course iteration, one can use data to decide upon the next improvements. Even in a good situation one can likely always improve – and, when processes for gathering educational data are in place, data can be given to external researchers. The feedback and results from external researchers can then again be used to increase the understanding of the current context, which potentially can be used again to create further actions – which again can be studied.

Next, we focus on the use of source code snapshots. First, we discuss the gathering and use of source code snapshot data. Then, a study where a manual analysis on the approaches that students take when writing their very first "Hello World" program is given (Publication III.1), followed by a study on the creation of automatically identifiable indicators of programming assignment difficulty (III.2). Much of the work described here is basic research, and many of the studies are experimental and have not been integrated to the everyday practice. However, information from the learning context has already been used to create interventions, which have shown promise (see e.g. [118]). For further studies where the data has been used, see e.g. [52, 68, 127].

5.1 Source-code Snapshot Data

For decades, there have been systems that record students' working processes while they work on programming assignments [54]. While at first these systems were used mainly for automated assessment and recorded only students' final submissions, systems that gather more than the end result have been developed later. These contemporary systems typically involve a software component that students need to install on their computer, and capture snapshots – *states in the source code as the student works towards a solution* – either using a defined interval or when students perform an action such as saving of the current work [54, 106].

Test My Code, which was described in Chapter 4, gives users the choice of providing data for researchers. In the most up-to-date version, every keypress within the programming environment can be recorded and stored with a timestamp. A recent study by us showed that such detailed information is beneficial as many students work on assignments that they never submit for grading, and also work on assignments after they have been submitted [116]. That is, when given the opportunity to investigate a fine-grained programming process, more insight into the students' challenges may be gained. When comparing Test My Code to other efforts in data gathering such as the BlackBox project [20], our approach is more fine-grained and directly bound to the course materials, whilst the BlackBox project has a more substantial user base.

A growing body of research that uses the programming process rather than the final submissions exists [49]; weaker students, when editing their programs, often delete larger code blocks (e.g. entire methods) and then possibly rewrite those from scratch [3], at-risk students can be identified to some extent [56, 86, 113, 124], challenging concepts and assignments can be identified with the goal of refining the material [92], and hints can be provided based on the students' state [57].

The main benefit of data-driven approaches that look at how students solve programming errors [56,124], how they schedule their time or whether they pay attention to code quality [113] is that they are directly based on programming behavior of a student and therefore can directly reflect changes in their learning and skills over time. Moreover, as the data is recorded from the students' normal learning activity – programming – instead of additional or external tests, there is no overhead of e.g. aptitude tests [123].

5.2 On the First "Hello World" Application

Although a growing body of research exists on the struggles that students face when writing code within educational environments (see e.g. [16, 21, 49]), little has been done on the issues that students encounter when learning to program within standard, off-the-shelf programming environments. This is somewhat surprising, as one of the implicit reasons for the creation of these educational environments is that the standard off-the-shelf programming environments are not sufficient.

In order to fill this gap, we analyzed how students who have explicitly stated that they have no previous programming experience write their first programs inside an industry-standard programming environment (Publication III.1). The analysis was conducted by qualitatively inspecting how students approach writing simple statements, and quantitatively inspecting the source code compilation statistics and the most typical errors they face.

The results from the analysis of two weeks of programming data show no indication that tertiary-level students could not start to learn to program directly within a standard programming environment – on the contrary, the analysis showed that the students made active use of the features provided by the programming environment such as shortcuts and automatic completion of source code. Second, an analysis of source code errors from the programming environment indicates that students’ struggles are different to those in novice programming environments such as BlueJ [63], where students need to run their applications to get feedback that indicates whether their code is in a compiling state or not.

As a consequence, this suggested that methods for identifying at-risk students that are based on data from novice programming environments (e.g. Jadud’s EQ [56]) may be context-specific[1], and that more work on context- and tool-specific methods is needed. Finally, four strategies of approaching the very first hello-world application were identified:

linear In the *linear* approach, students type the print command character by character from left to right. In most cases, this is followed by the IDE automatically adding parentheses and a semicolon at the end of the method name. Typos and other errors may occur as later discussed.

autocomplete Most IDEs support autocompletion. In addition to adding parentheses at the end of method calls, the IDE suggests completions for classes and objects. For example, when typing `System.` the IDE will open a popup with alternative completions. Typing more characters will limit the completions in the popup accordingly.

ide shortcuts such as `sout` As the print statement is commonly used but rather long in Java, IDEs offer shortcuts such as `sout` that, when

[1]This suggestion was subsequently studied by us in [78].

typed and followed by pressing tab, is automatically replaced with the print command `System.out.println("");`. In addition, the cursor jumps inside the quotes so that the user can immediately go on to type the contents of the string. The course material hints to the students to try the command after they have worked on a number of programming assignments.

copy-paste The last observed category of typing the print statement is to copy and paste code from somewhere else and then edit the argument only.

At the beginning of the course, many students copy-paste even simple commands such as printing. Those, who do write the commands themselves, do several types of mistakes during the process. These mistakes include:

1. Incorrectly mixing capital and small letter (e.g. `system` instead of `System` and `Out` instead of `out`)
2. Using other characters than the dot to access class members (e.g. `System-out-println`)
3. Various problems related to defining string literals, for example not using quotes at all (i.e. `System.out.println(Hello);`) or using parentheses instead of quotes (i.e. `System.out.println((Hello));`), and forgetting `.out` from the print command.
4. Normal typos without obvious misunderstanding behind them.

After a while – already during the second week of the course – the use of copy-pasting significantly reduces and students start to use shortcuts (e.g. `sout`). Similarly, the mistakes related to writing the simple print statement also almost disappear.

5.3 Feedback and Programming Assignment Difficulty

The difficulty of learning tasks is a major factor in learning, as is the feedback given to students. A teacher or educational environment can help a student reflect on her progress by providing feedback that relates the student's performance to a particular goal or subgoal. Feedback on progress should take into account the student's background and prior performance as well as the difficulty of the task; a beginner completing a difficult task may be applauded, but on the other hand, praising a student on success on a task that the student herself sees as trivial can be detrimental to self-efficacy and motivation [17]. Inappropriate feedback may also cause students to learn to distrust the feedback provider.

While automatic assessment systems bring benefits such as easy accessibility and low cost per student, the downside of many such systems is that they fall short of a human tutor in terms of quality of feedback. A part of this problem is that the feedback provided by automatic systems is typically not personalized to take the learner and the learner's present knowledge into account. In order to provide better automatic feedback, one needs to be able to judge how the individual learner relates to the assignment at hand. For example, does she find it difficult?

Task difficulty impacts students' motivation in several ways. For example, as per expectancy–value theories of motivation [5], assignments that are too easy are likely to have low perceived utility, while hard ones have a higher cost of completion, which reduces motivation unless they have been carefully designed to sustain interest. Excessive difficulty also contributes towards poor self-efficacy [8], which hampers further learning. Another form of scaffolding that impacts motivation is the feedback that learners receive. Hattie and Timperley [44] argue that the three main roles of feedback are to help a learner understand 1) the goals of learning, 2) the learner's own progress towards those goals, and 3) the activities that are needed to make better progress.

Research that seeks to improve the automatic assessment of students' solutions to programming problems exists [54]. Typically, automatic feedback is provided after students take an action such as submitting a solution; the feedback often consists of information on the correctness of the solution and perhaps some additional information about observed deficiencies. The feedback may also praise the student for getting a good score or exhort them to make an improved attempt. Two weaknesses of the typical approach discussed above are: 1) The feedback is "passive", as it is only presented when the student requests it, e.g., by submitting a solution, instead of being proactively offered, say, when the student is experiencing difficulty. 2) Feedback messages are based on the features of the submitted solution only, and are not influenced by other relevant factors such as the student's background or the difficulty of the task for the particular student. For instance, an experienced student may receive excessive accolades for a trivial assignment, which then undermines any praise received for more challenging ones.

When considering the snapshot data that Test My Code provides, it has the benefit that it is collected from the natural learning process and makes it possible to provide early, proactive feedback that the student does not need to explicitly request by submitting an assignment. Since, as Hattie and Timperley put it, “feedback is effective when it consists of information about progress, and/or about how to proceed” [44], such data has the potential to further enhance feedback.

In Publication III.2, we describe an exploration of indicators of programming assignment difficulty that can be automatically detected for each student from source code snapshots of the student’s evolving code, and then – in the future – used for crafting more effective feedback mechanisms. Initially, the analysis was carried out to determine the correlations between the programming assignment difficulty reported by students and individual factors including (1) the time spent, (2) the number of keystrokes, (3) the percentage of keystrokes and time in a non-compiling state, (4) the number of lines of code, and (5) the number of control-flow elements in the program (e.g. *if, else, while, for, return*). Line and control-flow element counts measure code complexity, and have previously been observed to be decent indicators of perceived difficulty [4].

While the majority of the observed correlations were statistically significant, the correlations were mostly medium-sized ($0.3 < r < 0.5$). In only two of the cases, the individual factors show a high correlation with difficulty ($r = 0.55; r = 0.53$); both factors being time. The pattern of correlations was largely similar for students with and without prior programming experience. After the analysis of individual factors, the factors were combined using a decision tree. The result indicated that although program size, complexity, and the degree to which the student maintained her program in a compilable state had an effect, students generally viewed time-consuming assignments as difficult.

The results show that metrics related to perceived difficulty can be automatically extracted from data that describes students’ programming process. This means that automatic feedback systems can be adjusted to take task difficulty into account, which may improve the quality of feedback of such systems in the future.

Chapter 6

Conclusions and Future Work

In this final chapter, we provide a brief overview of the work, and link the research questions to the results. We also discuss limitations of the study and outline some of the potential future research directions.

6.1 Revisiting the Research Questions

RQ1: How do existing teaching interventions proposed in the literature influence introductory programming course pass rates?

Almost every teaching intervention reported in the literature improved the course pass rates at those institutions where the interventions were studied (Publication I.1). At the same time, there was no "best practice" that everyone should adopt, and one could state that the most important thing for an educator is to try to improve. The results, however, are likely biased: it is likely that teaching interventions that lead to worse results are less likely to be published, be it due to the peer review process or to some other factors.

RQ2: What types of short- and long-term effects in students' course performance does the reorganization of an introductory programming course lead to?

We created and implemented a contemporary variant of the Cognitive Apprenticeship method for teaching programming to beginners (Publication I.2). The variant, called the Extreme Apprenticeship method, draws upon industry best practices and Extreme Programming, and adopts Cognitive Apprenticeship practices in the context of learning to program. The Extreme Apprenticeship method emphasizes meaningful practice and bi-directional feedback during the practice process, and scales to large amounts of learners. This scaling is partially realized by providing students with further learning opportunities via a deliberate apprenticeship phase, where students learn to coach other students as well as revisit their knowledge in the topics of the introductory programming courses (Publication I.3). For this to succeed, the teaching management must allow such an arrangement, and the structures must provide facilities that can be used for learning and coaching activities. Finally, the course instructors must provide meaningful learning materials, trust the learners, give learners responsibility, and support them – that is, the instructors need to be present and react when needed.

Overall, the observed effects are positive (Publication I.4). The reorganization led to a significant improvement in students' success during their early studies, and students are eager to participate in the teaching activities at the department. At the same time, students' success in their subsequent studies are influenced by their own choices in the subsequent courses – something that is out of the scope of this study.

RQ3: How do students admitted through a MOOC in programming perform in their studies when compared to traditionally admitted students?

We created a MOOC in programming that was used to admit students into the CS program at the University of Helsinki (Publication II.2). To support students in the MOOC, a tool called Test My Code that can be used to provide students with feedback during their programming process was constructed (Publication II.1). The students who have been admitted through the MOOC perform better during the first year than those students who have been admitted through the traditional path (Publication II.3). This does not mean, however, that the students that have been admitted through the MOOC are better in some way. Our results simply indicate that we are able to (1) bypass the introductory programming barrier that many face, and (2) identify students who are motivated to study computer science and willing to invest a few hundred hours even before being admitted to a University. If we would have compared only those students who have passed the introductory programming course at the University of Helsinki, the results could have been different – this is out of the scope of this work however.

RQ4: What do students who have never programmed before do when they take their first programming steps, and what types of factors contribute to the programming tasks feeling difficult?

In order to conduct such studies, Test My Code has the possibility to record and store students' programming process. The data from students who had identified themselves as novices revealed four distinct approaches for constructing their very first programs (Publication III.1). Writing programs linearly and copy-pasting parts of the content was to be expected. However, some of the students also relied on support from the programming environment, and used built-in autocompletion mechanisms and shortcuts. The study also explored the errors that students make as they are writing the programs and showed that errors related to writing basic commands disappeared rather early for most of the students. This effectively indicates that – at least in the context of the study – there is no explicit need for a separate programming environment for novices in which the more advanced functionalities are disabled.

Overall, when considering the difficulty of a programming assignment, the study showed that the time used on a task was one of the largest contributors to the observed difficulty (Publication III.2). This does not mean, however, that all activities that take time are seen as difficult. In the learning materials used, tasks are often organized so that they are in an order of increasing difficulty. Moreover, when a new topic or construct is studied, students first practice the topic with smaller assignments, and then work on a larger task in which they apply the constructs they have learned.

6.2 Closely-related Work

The work described here spans multiple research areas. It includes reorganizing the way introductory programming courses are taught by creating and adopting pedagogical alternatives, creating tools that support students as they are learning to program, creating new opportunities for entering university in addition to traditional entrance exams, and using students' learning process data to seek information that can be used for continuous improvement of the instruction provided. Due to the span over these multiple areas, the related work is scattered throughout this thesis.

Here, we seek to bring together the separate themes by reviewing related work, with comparison to our own work. Note that, unfortunately, as the field is vast and has been studied for decades, a larger review is out of the scope of this thesis. As work on the scale of this thesis is rarely reported, related work is visited in each of the research areas separately, starting with rethinking the way in which introductory programming courses are organized, and finishing with data-analytics and continuous improvement.

Rethinking introductory programming instruction

The notion of substituting lecture time with more meaningful activities has been thought about in the past. One of the closest approaches to ours, lab-centric instruction [110], reduces the role of lectures and uses lab sessions as the primary resource for learning. The labs "[have] activities to be accomplished during a scheduled time under the supervision of a member of the course staff". When compared to the Extreme Apprenticeship method, the learning activities in lab-centric instruction are more diverse, ranging from answering to quizzes to brainstorming and design tasks. While there are many similarities between lab-centric instruction and Extreme Apprenticeship, a number of differences do also exist – perhaps the most visible differences are in the emphasis and quantity of programming assignments, as well as in the way the labs are structured. In the context of Extreme Apprenticeship, there are literally hundreds of programming assignments, students who have just taken the course – junior advisors – participate in teaching, and the course participants can come and go as they will – no structured activities are used[1]. Benefits similar to those that have been observed with the Extreme Apprenticeship method, e.g. cost-effectiveness and feedback, have also been observed with lab-centric instruction [110].

The Cognitive Apprenticeship theory and the idea of apprentices have been used in the past in the context of introductory programming as well. One of the earlier discussions on tacit knowledge was written by Soloway in "*Learning to program = learning to construct mechanisms and explanations*" [101], where he analyzed students' solutions to a seemingly simple programming problem that required merging a number of programming constructs together. Soloway emphasized instructing students in the use of stepwise refinement when solving programming problems, and discussed strategies that can be used to compose these solutions together. These ideas can be seen as an operationalization of Cognitive Apprenticeship [77], and the need for existing knowledge for crafting solutions to new programming problems has also been one of the key reasons behind our emphasis on practical programming assignments.

Another approach has been described by Astrachan and Reed, who proposed the Applied Apprenticeship Approach [7]. It approaches learning to program by having students study and extend programs written by more experienced practitioners as "*students should be expected to read and modify programs before writing them*" [7]. One of the goals is to have

[1]At the time of the writing of this thesis, we have experimented with e.g. pair programming assignments, but due to the large number of students, no specific structures have been needed.

students learn design practices from this process, and eventually to learn to write such programs also from scratch. Astrachan and Reed also emphasize the need for relevant programming assignments, which in turn can show programming as a useful craft, as well as provide incremental examples of program design. Some of the ideas have been inspired by the work of Linn and Clancy [69], who describe the programs that students are to study as *case studies*, and also advocate on written explanations of the concrete design process – "*Students need more assistance than just expert code in order to learn design skills*" [69]. Their argument is based on a study where they compared populations where one population had access to expert commentary of program code, while another did not. The population that had access to the commentary fared better: "*even though the students examined the expert code and answered study questions intended to get them to infer the design process, this was not as effective as reading the expert commentary*" [69].

The same idea of starting by studying code written by experts has also been proposed more recently. In "*Enhancing Apprentice-Based Learning of Java*" [62], Kölling and Barnes propose the use of an educational programming environment called BlueJ [63] that helps students who are learning to program, and point out that in addition to being able to study code written by experts, students should also be able to observe the concrete working process. The observation is done in lectures, where an instructor demonstrates the steps needed to complete a specific software engineering task [62]. Many of these ideas have been collected to provide a starting point for an instructor who is planning to design a programming course by Caspersen and Bennedsen in "*Instructional Design of a Programming Course – A Learning Theoretic Approach*" [25].

When comparing these threads to our work, we have significantly deemphasized the need for having students read expert solutions, and have moved towards having students start with creating small programs from early on in the course. By working on dozens of such programs, also under guidance, the students can be coached towards desired outcomes and they are likely to also learn good working practices at the same time. While we have not emphasized the use of worked examples and having students study programs in our articles, tutorials that can be seen as case studies have also been embedded to the learning materials that students use. Moreover, in our approach, the role of feedback and the learning community is more significant, and the cost-effectiveness as well as the carry-on effect of the approach have also been assessed.

Tools that support students as they are learning to program

Although there are numerous systems that have been designed to help students learn programming [23], our focus here is on a narrow subset of these systems. We focus on systems that both assess students' programs, as well as provide guidance to students as they are learning to program by e.g. providing feedback based on a set of failed tests. While Test My Code was created to fulfill a specific pedagogical need [120], rapid feedback and support for rapid progress, many of the traditional systems have been created *system-first* to provide a full-fledged platform for instructors (see e.g. [24, 35]). The very first automated assessment system is from the early 1960's, where an automatic grader was first used in the context of learning to program – one of the baseline reasons for the creation of such a system was to be able to increase the amount of students admitted to a programming class [51]. Since the 1960s, numerous systems for automatically assessing students' programs have been designed [1, 33, 54, 87].

The systems closest to our work are most likely Web-CAT [35] and Marmoset [105], which have both been designed to improve the feedback cycle as a student is learning to program. While Web-CAT has been developed to support Test-Driven Development by allowing students to submit their own tests in addition to the assignment solutions which are then evaluated using both the code coverage of the students' own tests as well as tests written by an instructor, Marmoset puts additional focus on guiding students to live without instructor-written tests by providing means to give limited feedback before the deadline. As pointed out by Spacco, if students are given full test results, they may adopt the habit of *programming by 'Brownian motion', where students make a series of small, seemingly random changes to the code in the hopes of making their program pass the next test case* [105]. Moreover, while Marmoset can be used to gather fine-grained data from the students' programming process, Web-CAT mostly gathers the students' submissions.

In some aspects, such as providing the students with the ability to submit their own test cases, Test My Code is not as mature as it provides only limited functionality for assessing the test cases, while in other aspects, it may be more advanced. This is only natural, as the design choices and goals between the systems have been different. Our main design goal was to provide feedback and guidance as students are working on assignments with subgoals, and as the Extreme Apprenticeship method relies on dozens of weekly assignments, the process of retrieving and submitting the assignments was made as straightforward as possible.

Perhaps the most significant contribution of Test My Code is the ability to automatically download and submit programming assignments directly within the programming environment, which may reduce the amount of time that students spend on non-necessary tasks. At the same time, assessment feedback and guidance are also shown directly within the programming environment, and the creation of assignments, tests and feedback messages has been made straightforward. While both Web-CAT and Marmoset also provide a plugin for uploading assignments directly from a programming environment, in Test My Code the integration is tighter, and there is e.g. no need to specifically define the assignment that the student is about to submit. Due to this tighter integration, the data gathering facility provided by Test My Code can also gather more fine-grained data than Marmoset.

Even with the improvements, all of these systems still face the issue of adaptation. That is, while the use of smart learning systems has increased substantially during the last decades, issues with the adoption and dissemination of such tools and practices still exist [23]. For example, to our knowledge, although open source and generally available, Test My Code is currently only used by a handful of universities, colleges and schools.

Open online courses and university admission

To date, there has been plenty of research into MOOCs, but many of the articles are focused on more generic viewpoints such as participant demographics, satisfaction and behavior (see e.g. [61, 90]), stakeholder perspectives (see e.g. [34, 93]), as well as on the large scale data that can be gathered from MOOCs. MOOCs in programming have also gathered plenty of attention [39, 71].

Using a massive open online course (MOOC) for identifying students who are to be admitted into university studies is a novel idea, and to our understanding, the article "*Multi-faceted support for MOOC in programming*" [117], on which the included article II.2 "*A Purposeful MOOC to Alleviate Insufficient CS Education in Finnish Schools*" is built upon, is the first article on introductory programming MOOCs as a vehicle for university admission. While the work was performed to match a need, it is possible that such a need is not as prevalent in contexts in which students are expected to pay for tuition.

Perhaps the closest match to this work – while far-fetched – are studies on *programming aptitude*, where numerous factors that contribute to the ability of learning to program have been studied (see e.g. [37, 112]). However, while the open online course could in principle be seen as such a test, we do not consider it as such, and see it more as a way for students to see what studying Computer Science would include and to decide whether that would be something for them. At the same time, the course is also relatively far from advanced placement programs that are typical in the US (see e.g. [99]).

Data analytics and continuous improvement

As pointed out in Chapter 4, a number of tools for gathering data from students' learning process have been developed. Our contribution to such tools is Test My Code, which provides both the capability to scaffold students within the learning environment as well as gathers fine-grained information on the students' learning process. The data is tightly woven into the assignments that the student is working on, providing a roadmap of learning starting from the very first time that the student attempts programming.

In the data analytics part of this thesis, we have focused on two small cases where source code snapshot data has been used. The first case described an analysis of struggles that a novice programmer faces when learning a programming language within a standard programming environment. To our knowledge, this study is the first one where fine-grained programming behavior of novices has been observed within a programming environment. The study provides an initial demystification of standard off-the shelf programming environments that have traditionally been seen as too complex to use for novices. Naturally, such arguments are already almost a decade old (see e.g. [62]), so they may be based on observations that no longer would hold.

Although no direct matches to the first study exist, there are studies that analyze source code snapshots but with different granularities and environments. For example, in "Using CodeBrowser to seek differences between novice programmers" [48], we studied students' struggles in a number of early programming assignments in a programming course. In the study however, the granularity was more coarse and the number of participants smaller. Another contribution towards the understanding of students' problem solving strategies was performed by Blikstein [16], who studied fine-grained programming behavior of nine students learning to program within the NetLogo environment. More recently, there has been a movement towards browser-based learning environments, which have also been studied. For example, the work by Helminen et al. [49] discusses patterns that third-year students exhibit when learning to program using an interactive Python environment, and provides a good overview on work with source code snapshots.

Another stream of source code snapshot analysis is related to identifying students who struggle, in order to provide guidance for them. One of the first approaches to quantify students' tendency to create and fix errors was called the *error quotient*, which was calculated from subsequent source code snapshots. The approach has mediocre correlation with course outcomes [56]. Naturally the context should also be taken into account. More recently, an improvement on the error quotient was proposed by Watson et al., who noticed that the amount of time that students spend on programming assignments is also an important factor in determining the struggling students [124]. They proposed an improvement to the error quotient algorithm, called *Watwin*-algorithm, which had an improved predictive power over the error quotient in their context. Snapshots have been used to elicit information on a finer detail as well. For example, Piech et al. [79] studied students' approaches to solving a programming task using Karel the Robot, and found that the clusters created from the programming patterns were indicative of course midterm scores.

The data from Test My Code has also been studied in other contexts, for example Hosseini et al. have identified students' behaviors within a programming course – some students were more inclined to build their code step by step, while others started from larger quantities of code, and reduced their code in order to reach a solution [52]. Another approach recently proposed by Yudelson et al. is to use fine-grained concepts extracted from source code snapshots, and to model students' understanding of these concepts as they proceed within a course based on the source code snapshots [127].

The second case was related to identifying difficult programming assignments already during the students' working process for the purpose of providing more targeted guidance. While the importance of such knowledge has been highlighted also in the past (see e.g. [16]), the closest study on determining the difficulty of programming assignments has been performed by Alvares and Scott [4]. While the difference in results are likely explainable with the number of participants and data gathering methodology, it is evident that such information can be gathered and taken into use. In our work, the results were based on data from Test My Code, which means that guidance could potentially be given to the students within the working environment already before they submit their work.

6.3 Limitations of the Work

Next, we outline limitations of the work and address both internal and external validity concerns. Internal validity refers to how well the studies have been conducted, and whether it is possible that there are other causes for the outcomes of the research. External validity, on the other hand, refers to whether the results generalize to other contexts as well. Internal and external validity concerns for each theme are considered separately.

Theme 1: Retention in Introductory Programming Courses and the Extreme Apprenticeship method

When considering the survey of existing work in teaching programming (Publication I.1), one of the concerns with the work is that we excluded a range of articles that did not report the teaching practices or the results in a sufficient manner. It is possible that when relaxing the inclusion criteria, the results could change due to this. Moreover, it is likely that the field favors positive results, which means that there are likely several studies with negative results that have never been published.

When considering internal and external validity concerns of the Extreme Apprenticeship method (Publications I.2, I.3), there are multiple concerns. First, we did not use control and treatment groups, nor did we use pre- and post-tests for assessing students' knowledge. Similarly, while the exams before and after the transition contained many identical components, a manual analysis by a third party indicated that the exams used after the transition expect and assess a wider range of knowledge.

Moreover, our study did not attempt to tease out the individual effects of the components of the Extreme Apprenticeship method. That is, our results come from using a combination of components, but we do not know the individual effect of, for example, starting early, having smaller tasks, having larger tasks that integrate knowledge from the smaller tasks, having help available, using worked examples, using authentic working environments and so on. That is, it is possible that some of the components of the Extreme Apprenticeship method may not be beneficial for the student.

When considering internal and external validity concerns of the so-called "Carry-on Effect" of the Extreme Apprenticeship method (Publication I.4), we acknowledge that the drive for improvement at the University of Helsinki was not limited to the introductory programming courses. Other instructors may have sought to improve their courses, which may have an overall net effect on the observed mentality at the Department of Computer Science, which in turn may have influenced the results. That is, it is not clear if the observed effect was due to the Extreme Apprenticeship method or if there are other factors that contributed to the outcomes. Moreover, the study included data from year 2012, where the student population also had students who had been admitted through the MOOC. Nevertheless, the analysis was conducted on the student population as a whole.

Overall, it is possible that some of the results could be explained by population bias. Instructors play a role and, as the author of this thesis has participated in teaching the studied courses, it is possible that taking only the methodology to another context would not yield similar results. There are, however, positive results of the method from other contexts (see e.g. [32, 46, 84]). Applicability of this work in other contexts still remains an open question – research on the methods proposed in this thesis has not yet answered the questions of when they work, where they work, and most importantly, when they do not work.

In our context, students have a right to study and they do not pay tuition. In addition, students receive a number of benefits such as monthly monetary support from the government, discounts from public traffic tickets, and opportunities for low-cost housing. Such factors can have a large influence. Similarly, the practicalities involved with empowering first-year students and taking them as a part of the teaching staff may be a challenge in some contexts. For this thesis, the administration allowed us to completely change how the first introductory programming courses are organized, which may not be as straightforward in other contexts.

Theme 2: Distance Education and a MOOC Focused on Programming

When considering the internal and external validity concerns of the Theme 2, we must first note that the Test My Code system (Publication II.1) has not been formally evaluated. We do not and cannot claim that Test My Code would have improved students' learning gains. Nevertheless, without the system (or another similar system), offering the introductory programming courses as a MOOC would not have been possible. We have constructed a MOOC that can be taken by anyone and offered it to Finnish schools free of charge (Publication II.2), but we have not studied the extent to which the course actually has been adapted in Finnish schools.

When considering the internal validity of our study of students who have been admitted to study Computer Science through the MOOC in Programming, the preliminary study included in this thesis (Publication II.3) reported study success of different student cohorts. At the time of the writing of the article, the goal was to report and show that this can be done and to provide preliminary results. We deliberately chose to not conduct tests for statistical significance as we thought that any results would be preliminary. We acknowledge that this can be seen as a poor choice, but also note that statistical tests can be calculated based on the data given in the reported tables.

Moreover, we only studied the initial student population, and would need subsequent studies that analyze the way the cohorts succeed in their studies over the years. Conducting such a study is challenging, however, as a range of factors contribute to students' choices of continuing or not to continuing to study.

When considering the external validity of our work, we do not know if the approach would generalize to other contexts. It is possible, for example, that other universities would not have similar pull as ours. Similarly, if all universities would open a similar admission pathway, we do not know what would happen. Our preliminary experiences, however, indicate that the populace in the MOOC admission path and the traditional admission path do not completely overlap, which means that at least considering multiple admission paths is likely beneficial.

Theme 3: Mining Student Data

When considering the articles that focused on mining student data and the internal and external validity of those studies, multiple issues are present. In the article in which we studied how students write their very first lines of code (Publication III.1), one internal validity concern is related to the way the manual encoding was conducted. Instead of conducting the encoding separately and using e.g. Cohen's kappa to assess the encoding quality, we analyzed the results together. While unlikely, it is still possible that other researchers would have come up with different encoding.

Similarly, when considering external validity, we do not know if the way students type their first programs presented in the article generalize to other contexts. In the article, we compared other metrics such as typical compilation errors to other relevant studies. We do not, however, know if the reason why the results differ are actually due to the programming environment, or if other factors contribute to the results – in other terms, we did not conduct a control study in which one population was given a specific programming environment and another population was given another programming environment.

When considering the internal and external validity of the article in which we sought for automatic ways to determine programming assignment difficulty (Publication III.2), the results are only as valid as the methodologies used. One of the primary internal validity concerns in the work is related to the notion of difficulty; we do not know if the students perceived the question on how difficult a programming assignment was in the way that we intended the question. It is possible, for example, that difficulty was at least partially mixed with perceived workload of the assignments. Similarly, as we only received feedback from those students who completed assignments, there is a possibility for selection bias in the data.

So far, we cannot state if the same results would apply in other contexts. Thus, the external validity of the results are still an open question.

6.4 Future Directions

While in addition to programming the Extreme Apprenticeship method has already been studied in teaching e.g. operating systems [32] and mathematics [46, 84], further studies of such issues as the practice, students' views, as well as learning gains are needed. From the perspective of a researcher, these studies should be randomized controlled trials where the student population is split e.g. into treatment and control groups, while from the perspective of an educator, creating such a study – especially if there is prior belief on one method working better than the other – can be perceived unethical for the students who have to study with the "worse" method. Even with such concerns, numerous research directions exist. For example, what are the long-term effects of having students participate as educators already during their first year? What attributes do students value from the peers who assist them? Can students' actions and behavior be guided towards such attributes? Even studying the components needed in useful learning materials has a number of open questions, some of which we have only recently scratched the surface of [43, 118].

The strategy of reducing the number of lectures to increase the amount of productive work can also be further studied. While the change has led to an increase in students who succeed in their first year studies, students' motivations and the driving factors of the method can be studied further. For example, what is the effect of the step-wise assignments that students work on in order to learn the relevant working practices, and what is the effect of e.g. pseudo-external factors such as course grading schemes? Plenty of work that outlines the value of peer instruction, pair programming and media computation also exists [81], and adopting practices such as pair programming into the lab would be interesting to study e.g. from the perspective of a junior advisor.

The tools used to create and enable the open online course also warrant further study. As we pointed out in the article on automatically detectable indicators of programming assignment difficulty [55], the feedback that is given to the students during the programming and learning process is important. Automatic assessment systems with better personalized feedback can be created to support students who would rather work from home, but one can ask whether that direction is the correct way to go. As the driving factors of Extreme Apprenticeship include the bi-directional feedback and the ability to work with students locally, as well as having the junior advisors teach others, it is questionable if efforts that effectively help students stay at home are beneficial – even if it may be more comfortable for them.

Similarly, while the amount of guidance and feedback such systems should provide is an open question, there exists numerous paths of study that are related to such learning environments. For example, what are the long-term effects of having students follow specific practices such as coding conventions – do they continue to use them when they are no longer required by the environment? This could already be studied e.g. by performing a post-hoc analysis of student programming projects, say, from years 2007-09 and 2010-12.

As Test My Code continuously collects data from students' programming process, the data also provides ample opportunities for additional analysis and support. Building estimates on the students' skill-level in order to personalize their learning experience is mostly untapped – especially in programming courses with hundreds of programming assignments. Similarly, new opportunities for understanding the plans that students formulate when solving a programming problems arise, and perhaps such studies could even provide further insight on how we learn crafts such as programming. Questions on the differences between novices and professionals could also already be answered since we have gathered students' programming background details already for the past few years.

And, now that we touched the topic of professional programmers, it would be interesting to study whether the methodologies and tools outlined in this thesis would be sensible to use when introducing professionals into new domains.

References

[1] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] Satu Alaoutinen. *Enabling constructive alignment in programming instruction*. PhD thesis, Lappeenranta University of Technology, 2011.

[3] Anthony Allevato and Stephen H Edwards. Discovering patterns in student activity on programming assignments. In *2010 ASEE Southeastern Section Annual Conference and Meeting*, 2010.

[4] Andres Alvarez and Terry A. Scott. Using student surveys in determining the difficulty of programming assignments. *Journal of Computing Sciences in Colleges*, 26(2):157–163, December 2010.

[5] Eric M. Anderman and Heather Dawson. Learning with motivation. Routledge, 2011.

[6] Terry Anderson. Getting the mix right again: An updated and theoretical rationale for interaction. *The International Review of Research in Open and Distance Learning*, 4(2), 2003.

[7] Owen Astrachan and David Reed. AAA and CS 1: The applied apprenticeship approach to CS 1. In *ACM SIGCSE Bulletin*, volume 27, pages 1–5. ACM, 1995.

[8] Albert Bandura. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review*, 84(2):191, 1977.

[9] Sasha Barab and Kurt Squire. Design-based research: Putting a stake in the ground. *Journal of the Learning Sciences*, 13(1):1–14, 2004.

[10] Jessica D Bayliss. The effects of games in CS1-3. In *Microsoft Academic Days Conference on Game Development in Computer Science Education*, pages 59–63. Citeseer, 2007.

[11] Jessica D. Bayliss and Sean Strout. Games as a "flavor" of CS1. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, pages 500–504. ACM, 2006.

[12] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[13] Jens Bennedsen and Michael E Caspersen. Failure rates in introductory programming. *SIGCSE Bulletin*, 39(2):32–36, 2007.

[14] Eric P Bettinger and Bridget Terry Long. Does cheaper mean better? The impact of using adjunct instructors on student outcomes. *The Review of Economics and Statistics*, 92(3):598–613, 2010.

[15] Toni R. Black. Helping novice programming students succeed. *Journal of Computing Sciences in Colleges*, 22(2):109–114, 2006.

[16] Paulo Blikstein. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pages 110–116, New York, NY, USA, 2011. ACM.

[17] Gary D. Borich and Martin L. Tombari. *Educational Psychology: A Contemporary Approach*. Longman Publishing/Addison Wesley, 2nd edition, 1997.

[18] David Boud, Rosemary Keogh, David Walker, et al. *Reflection: Turning experience into learning*. Routledge, 1985.

[19] John Seely Brown, Allan Collins, and Paul Duguid. Situated cognition and the culture of learning. *Educational researcher*, 18(1):32–42, 1989.

[20] Neil C. C. Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 223–228, New York, NY, USA, 2014. ACM.

[21] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[22] Jerome Bruner. Vygotsky: A historical and conceptual perspective. *Culture, communication, and cognition: Vygotskian perspectives*, pages 21–34, 1985.

[23] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, Jaime Urquiza, Arto Vihavainen, and Michael Wollowski. Increasing adoption of smart learning content for computer science education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*, ITiCSE-WGR '14, pages 31–57, New York, NY, USA, 2014. ACM.

[24] Peter Brusilovsky, Elmar Schwarz, and Gerhard Weber. ELM-ART: An intelligent tutoring system on World Wide Web. In *Intelligent tutoring systems*, pages 261–269. Springer, 1996.

[25] Michael E. Caspersen and Jens Bennedsen. Instructional design of a programming course: A learning theoretic approach. In *Proceedings of the Third International Workshop on Computing Education Research*, ICER '07, pages 111–122, New York, NY, USA, 2007. ACM.

[26] J. D. Chase and Edward G. Okie. Combining cooperative learning and peer instruction in introductory computer science. *SIGCSE Bull.*, 32(1):372–376, March 2000.

[27] Allan Collins, John Seely Brown, and Ann Holum. Cognitive apprenticeship: making thinking visible. *American Educator*, 6:38–46, 1991.

[28] Allan Collins, John Seely Brown, and Susan E. Newman. Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In *Knowing, Learning and Instruction: Essays in honor of Robert Glaser*. Hillside, 1989.

[29] Allan Collins and James G. Greeno. Situative view of learning. In Vibeke G. Aukrust, editor, *Learning and Cognition*, pages 64–68. Elsevier Science, 2010.

[30] Noe De La Mora and Christine F Reilly. The impact of real-world topic labs on student performance in CS1. In *2012 Frontiers in Education Conference Proceedings*, pages 1–6. IEEE, 2012.

[31] Paul E Dickson. Using undergraduate teaching assistants in a small college environment. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 75–80. ACM, 2011.

[32] Gabriella Dodero and Francesco Di Cerbo. Extreme apprenticeship goes blended: An experience. *Advanced Learning Technologies, IEEE International Conference on*, 0:324–326, 2012.

[33] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3), September 2005.

[34] Anna Eckerdal, Päivi Kinnunen, Neena Thota, Aletta Nylén, Judy Sheard, and Lauri Malmi. Teaching and learning with MOOCs: Computing academics' perspectives and engagement. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 9–14, New York, NY, USA, 2014. ACM.

[35] Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 148–155, New York, NY, USA, 2003. ACM.

[36] K. Anders Ericsson, Ralf T. Krampe, and Clemens Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3):363, 1993.

[37] Gerald E. Evans and Mark G. Simkin. What best predicts computer proficiency? *Communications of the ACM*, 32(11):1322–1327, November 1989.

[38] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182 –211, 1976.

[39] Katrina Falkner, Nickolas Falkner, Claudia Szabo, and Rebecca Vivian. Applying validated pedagogy to MOOCs: An introductory programming course with media computation. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 326–331, New York, NY, USA, 2016. ACM.

[40] S. Fincher, B. Richards, J. Finlay, H. Sharp, and I. Falconer. Stories of change: How educators change their practice. In *Frontiers in Education Conference (FIE), 2012*, pages 1–6, 2012.

[41] James G Greeno. Response: On claims that answer the wrong questions. *Educational Researcher*, 26(1):5–17, 1997.

[42] Frank M. Grittner. Individualized instruction: An historical perspective. *The Modern Language Journal*, 59(7):323–333, 1975.

[43] Lassi Haaranen, Petri Ihantola, Juha Sorva, and Arto Vihavainen. In search of the emotional design effect in programming. In *Joint Software Engineering Education and Training track of the 37th International Conference on Software Engineering*, JSEET/ICSE '15, 2015.

[44] John Hattie and Helen Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[45] Michael Haungs, Christopher Clark, John Clements, and David Janzen. Improving first-year success and retention through interest-based CS0 courses. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 589–594, New York, NY, USA, 2012. ACM.

[46] Terhi Hautala, Tiina Romu, Johanna Rämö, and Thomas Vikberg. Extreme apprenticeship method in teaching university-level mathematics. In *Proceedings of the 12th International Congress on Mathematical Education, ICME*, 2012.

[47] Anja Heikkinen and Ronald G Sultana. *Vocational Education and Apprenticeships in Europe. Challenges for Practice and Research. University of Tampere Department of Education Series B, No. 16.* ERIC, 1997.

[48] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 229–234. ACM, 2014.

[49] Juha Helminen, Petri Ihantola, and Ville Karavirta. Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 13–22, New York, NY, USA, 2013. ACM.

[50] Juha Helminen and Lauri Malmi. Jype - a program visualization and programming exercise tool for Python. In *Proceedings of the 5th*

International Symposium on Software Visualization, SOFTVIS '10, pages 153–162, New York, NY, USA, 2010. ACM.

[51] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, October 1960.

[52] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. Exploring problem solving paths in a Java programming course. *Psychology of Programming Interest Group Annual Conference 2014*, 2014.

[53] Petri Ihantola. *Automated assessment of programming assignments: visual feedback, assignment mobility, and assessment of students' testing skills.* PhD thesis, Aalto University, 2011.

[54] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93. ACM, 2010.

[55] Petri Ihantola, Juha Sorva, and Arto Vihavainen. Automatically detectable indicators of programming assignment difficulty. In *Proceedings of the 15th Annual Conference on Information Technology Education*, SIGITE '14, pages 33–38, New York, NY, USA, 2014. ACM.

[56] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84, 2006.

[57] Wei Jin, Tiffany Barnes, John Stamper, Michael John Eagle, Matthew W. Johnson, and Lorrie Lehmann. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems*, ITS'12, pages 304–309, Berlin, Heidelberg, 2012. Springer-Verlag.

[58] Erkki Kaila, Teemu Rajala, Mikko-Jussi Laakso, Rolf Lindén, Einari Kurvinen, Ville Karavirta, and Tapio Salakoski. Comparing student performance between traditional and technologically enhanced programming course. In *Proceedings of the 17th Australasian Computing Education Conference*, pages 147–154, 2015.

[59] Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen. Carry-on effect in extreme apprenticeship. In *Frontiers in Education Conference, 2013 IEEE*, pages 1150–1155. IEEE, 2013.

[60] Päivi Kinnunen, Maija Marttila-Kontio, and Erkki Pesonen. Getting to know computer science freshmen. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 59–66, New York, NY, USA, 2013. ACM.

[61] René F. Kizilcec, Chris Piech, and Emily Schneider. Deconstructing disengagement: Analyzing learner subpopulations in massive open online courses. In *Proceedings of the Third International Conference on Learning Analytics and Knowledge*, LAK '13, pages 170–179, New York, NY, USA, 2013. ACM.

[62] Michael Kölling and David J. Barnes. Enhancing apprentice-based learning of Java. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 286–290, New York, NY, USA, 2004. ACM.

[63] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[64] Jaakko Kurhila and Arto Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, pages 3–8. ACM, 2011.

[65] Patricia Lasserre and Carolyn Szostak. Effects of team-based learning on a CS1 course. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 133–137, New York, NY, USA, 2011. ACM.

[66] Jean Lave and Etienne Wenger. *Situated learning: Legitimate peripheral participation.* Cambridge university press, 1991.

[67] Jari Lavonen and Seppo Laaksonen. Context of teaching and learning school science in Finland: Reflections on PISA 2006 results. *Journal of Research in Science Teaching*, 46(8):922–944, 2009.

[68] Marianne Leinikka, Arto Vihavainen, Jani Lukander, and Satu Pakarinen. Cognitive flexibility and programming performance. *Psychology of Programming Interest Group Annual Conference 2014*, 2014.

[69] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

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

[71] Heather Miller, Philipp Haller, Lukas Rytz, and Martin Odersky. Functional programming for all! Scaling a MOOC for students and professionals alike. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 256–263. ACM, 2014.

[72] Paul Mullins, Deborah Whitfield, and Michael Conlon. Using Alice 2.0 as a first language. *Journal of Computing Sciences in Colleges*, 24(3):136–143, 2009.

[73] Thomas L Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, et al. Exploring the role of visualization and engagement in computer science education. In *ACM SIGCSE Bulletin*, volume 35, pages 131–152. ACM, 2002.

[74] Uolevi Nikula, Orlena Gotel, and Jussi Kasurinen. A motivation guided holistic rehabilitation of the first programming course. *ACM Transactions on Computing Education*, 11(4):24:1–24:38, November 2011.

[75] Peter Norvig. Teach yourself programming in ten years. http://norvig.com/21-days.html. Accessed: 2016-04-01.

[76] Arnold Pears, Stephen Seidman, Crystal Eney, Päivi Kinnunen, and Lauri Malmi. Constructing a core literature for computing education research. *SIGCSE Bulletin*, 37(4):152–161, December 2005.

[77] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In

ITiCSE Working Group Reports, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.

[78] Andrew Petersen, Jaime Spacco, and Arto Vihavainen. An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 77–86, New York, NY, USA, 2015. ACM.

[79] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[80] Kian L. Pokorny. What will they know? Standards in the high school computer science curriculum. *Journal of Computing Sciences in Colleges*, 28(5):218–225, May 2013.

[81] Leo Porter and Beth Simon. Retaining nearly one-third more majors with a trio of instructional best practices in CS1. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 165–170, New York, NY, USA, 2013. ACM.

[82] Danijel Radošević, Tihomir Orehovački, and Alen Lovrenčić. New approaches and tools in teaching programming. In *Proceedings of Central European Conference on Information and Intelligent Systems*, pages 49–57, 2009.

[83] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. Effectiveness of program visualization: A case study with the ViLLE tool. *Journal of Information Technology Education*, 7, 2008.

[84] Johanna Rämö and Thomas Vikberg. Extreme apprenticeship–engaging undergraduate students on a mathematics course. In *Proceedings of the Frontiers in Mathematics and Science Education Research Conference*, FISER'14, 2014.

[85] Mona Rizvi and Thorna Humphries. A scratch-based CS0 course for at-risk computer science majors. In *2012 Frontiers in Education Conference Proceedings.*

[86] Ma Mercedes T Rodrigo, Ryan S Baker, Matthew C Jadud, Anna Christine M Amarra, Thomas Dy, Maria Beatriz V Espejo-Lahoz, Sheryl Ann L Lim, Sheila AMS Pascua, Jessica O Sugay, and Emily S

Tabanao. Affective and behavioral predictors of novice programmer achievement. *ACM SIGCSE Bulletin*, 41(3):156–160, 2009.

[87] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 133–136, New York, NY, USA, 2001. ACM.

[88] Robert Keith Sawyer. *The Cambridge handbook of the learning sciences.* Cambridge University Press New York, 2006.

[89] AH Schoenfeld. Bridging the cultures of educational research and design. *Educational Designer*, 1(2), 2009.

[90] Daniel T. Seaton, Yoav Bergner, Isaac Chuang, Piotr Mitros, and David E. Pritchard. Who does what in a massive open online course? *Communications of the ACM*, 57(4):58–65, April 2014.

[91] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto 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.

[92] Steven C. Shaffer and Mary Beth Rosson. Increasing student success by modifying course delivery based on student submission data. *ACM Inroads*, 4(4):81–86, December 2013.

[93] Judy Sheard, Anna Eckerdal, Päivi Kinnunen, Lauri Malmi, Aletta Nylén, and Neena Thota. MOOCs and their impact on academics. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 137–145, New York, NY, USA, 2014. ACM.

[94] Jim Shore. Fail fast. *IEEE Software*, 21(5):21–25, 2004.

[95] Simon. *Emergence of computing education as a research discipline.* PhD thesis, Aalto University, 2015.

[96] Beth Simon, Päivi Kinnunen, Leo Porter, and Dov Zazkis. Experience report: CS1 for majors with media computation. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, pages 214–218. ACM, 2010.

[97] Robert H. Sloan and Patrick Troy. CS 0.5: A better approach to introductory computer science for majors. *SIGCSE Bulletin*, 40(1):271–275, March 2008.

[98] Finbarr Sloane. Normal and design sciences in education: Why both are necessary. *Educational design research*, pages 19–44, 2006.

[99] Lawrence Snyder, Tiffany Barnes, Dan Garcia, Jody Paul, and Beth Simon. The first five computer science principles pilots: Summary and comparisons. *ACM Inroads*, 3(2):54–57, June 2012.

[100] David L Soldan, William P Osborne, and Don Gruenbacher. Modeling the economic cost of inadequate teaching and mentoring. In *Frontiers in Education Conference (FIE), 2010 IEEE*, pages F3J–1. IEEE, 2010.

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

[102] Juha Sorva. *Visual Program Simulation in Introductory Programming Education*. PhD thesis, Aalto University, 2012.

[103] Juha Sorva, Ville Karavirta, and Lauri Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):15:1–15:64, November 2013.

[104] Juha Sorva and Otto Seppälä. Research-based design of the first weeks of CS1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 71–80, New York, NY, USA, 2014. ACM.

[105] Jaime Spacco. *Marmoset: A programming project assignment framework to improve the feedback cycle for students, faculty and researchers*. PhD thesis, University of Maryland, 2006.

[106] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. *SIGCSE Bulletin*, 38(3):13–17, June 2006.

[107] Carolee Stewart-Gardiner. Improving the student success and retention of under achiever students in introductory computer science. *Journal of Computing Sciences in Colleges*, 26(6):16–22, 2011.

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

[109] Allison Elliott Tew, Charles Fowler, and Mark Guzdial. Tracking an innovation in introductory CS education from a research university to a two-year college. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 416–420. ACM, 2005.

[110] Nathaniel Titterton, Colleen M Lewis, and Michael J Clancy. Experiences with lab-centric instruction. *Computer Science Education*, 20(2):79–102, 2010.

[111] Allen B. Tucker. K-12 computer science: Aspirations, realities, and challenges. In *Proceedings of the 4th International Conference on Informatics in Secondary Schools - Evolution and Perspectives: Teaching Fundamentals Concepts of Informatics*, ISSEP '10, pages 22–34, Berlin, Heidelberg, 2010. Springer-Verlag.

[112] Markku Tukiainen and Eero Mönkkönen. Programming aptitude testing as a prediction of learning to program. *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*, pages 45–57, 2002.

[113] Arto Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Proceedings of the 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 498–499, 2013.

[114] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[115] Arto Vihavainen and Matti Luukkainen. Results from a three-year transition to the extreme apprenticeship method. *Proceedings of the 13th IEEE International Conference on Advanced Learning Technologies*, July 2013.

[116] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information Technology Education*, SIGITE '14, pages 21–26, New York, NY, USA, 2014. ACM.

[117] Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 171–176, New York, NY, USA, 2012. ACM.

[118] Arto Vihavainen, Craig S. Miller, and Amber Settle. Benefits of self-explanation in introductory programming. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 284–289, New York, NY, USA, 2015. ACM.

[119] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Jaakko Kurhila. Massive increase in eager TAs: Experiences from extreme apprenticeship-based CS1. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, ITiCSE '13, pages 123–128, New York, NY, USA, 2013. ACM.

[120] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, 2013.

[121] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes.* Harvard University Press, Cambridge, MA, 1978.

[122] Henry M Walker. Collaborative learning: a case study for CS1 at grinnell college and austin. In *SIGCSE Bulletin*, volume 29, pages 209–213. ACM, 1997.

[123] Christopher Watson and Frederick W.B. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pages 39–44, New York, NY, USA, 2014. ACM.

[124] Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 319–323, 2013.

[125] Paula Wilcox, Sandra Winn, and Marylynn Fyvie-Gauld. 'It was nothing to do with the university, it was just the people': the role of social support in the first-year experience of higher education. *Studies in higher education*, 30(6):707–722, 2005.

[126] Laurie Williams, Charlie McDowell, Nachiappan Nagappan, Julian Fernald, and Linda Werner. Building pair programming knowledge through a family of experiments. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ISESE '03, Washington, DC, USA, 2003. IEEE Computer Society.

[127] Michael Yudelson, Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. Investigating automated student modeling in a Java MOOC. *Proceedings of The Seventh International Conference on Educational Data Mining 2014*, 2014.

Publication I.1

Arto Vihavainen, Jonne Airaksinen, and Christopher Watson

A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success

In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER '14)*

A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success

Arto Vihavainen and Jonne Airaksinen
Department of Computer Science
University of Helsinki
Finland
{ avihavai, jonnaira }@cs.helsinki.fi

Christopher Watson
School of Engineering and Computing Sciences
University of Durham
United Kingdom
christopher.watson@dunelm.org.uk

ABSTRACT

Decades of effort has been put into decreasing the high failure rates of introductory programming courses. Whilst numerous studies suggest approaches that provide effective means of teaching programming, to date, no study has attempted to quantitatively compare the impact that different approaches have had on the pass rates of programming courses. In this article, we report the results of a systematic review on articles describing introductory programming teaching approaches, and provide an analysis of the effect that various interventions can have on the pass rates of introductory programming courses. A total of 60 pre-intervention and post-intervention pass rates, describing thirteen different teaching approaches were extracted from relevant articles and analyzed. The results showed that on average, teaching interventions can improve programming pass rates by nearly one third when compared to a traditional lecture and lab based approach.

Categories and Subject Descriptors

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

General Terms

Human Factors

Keywords

programming education, introductory programming, cs1, teaching interventions, analysis, systematic review

1. INTRODUCTION

The mean worldwide failure rates of CS1 have been suggested to be as high as one third of students failing the course [3]. A recent study showed that despite advances in pedagogy, the worldwide failure rates of CS1 have not improved over time, and that the failure rates are not substantially influenced by aspects of the external teaching context, such as the programming language taught in the course [25].

Despite decades of research, internal factors based upon traditional learning theories have also failed to explain the CS1 failure rate phenomenon, and no factor to date has been shown to influence programming performance across a range of different teaching contexts [27]. More recently, researchers have explored the relations between desirable aspects of programming behaviour and performance [23, 26]. But despite yielded promising results there is still no overall understanding as to why many students are able to program, whilst others endlessly struggle.

Many theories have been put forward as to why learning to program is a difficult task. Some are attributed to the nature of programming itself. Programming is not a single skill, but rather a complex cognitive activity, where a student must simultaneously build and apply several higher order cognitive skills to solve a particular problem [17]. Other reasons are attributed to aspects of the students. Students may lack motivation, they may be unable to create a mental model of how programs relate to the underlying system [2], or create a clear model of program flow [10].

Further reasons are associated with the teaching methodology used. Whilst many students fail programming courses, instructors face the additional challenge of adjusting their expectations to the students' level of ability [22]. These challenges have been acknowledged, and decades of research effort has been put into creating and applying teaching interventions that facilitate students' learning [17, 13]. These interventions can include moving from a traditional lecture and lab based approach to using pair programming, game based learning, or extreme apprenticeship.

However, to date, no study has attempted to quantitatively compare the impact that these approaches can have on improving the pass rates of failing programming courses. Without any quantitative evidence on the relative strengths of different approaches, the research community as a whole will continue to lack a clear consensus of precisely which methodologies provide the most effective means of teaching programming and saving failing programming students.

2. RESEARCH METHOD

The purpose of this study was to explore the degree to which various approaches of teaching programming could improve pass rates. In order to gather data for use in this study, a systematic review process was adopted, in an effort to identify as many articles that provided details of pre-intervention, and post-intervention pass rates as possible.

ICER '14 August 11 - 13 2014, Glasgow, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2755-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2632320.2632349.

2.1 Research Questions

The questions answered in this study include:

1. How do teaching interventions reported in the literature increase students' success in CS1?
2. What practices do the successful teaching interventions comprise of?
3. Do so called best practices, or practices that are significantly better than others exist?

2.2 Identification of Relevant Literature

To identify approaches for teaching introductory programming and their influences on course success, an initial search of articles published between the years 1980 - January 2014 was carried out. Searches were made in the ACM and IEEE databases, after which further searches were made using Google Scholar in an attempt to identify both published and unpublished work which was not indexed by ACM and IEEE. A final search was conducted by manually screening the indexes of selected conference proceedings and journals for relevant studies, including: (1) Transactions on Computing Education, (2) SIGCSE, (3) ITiCSE, (4) ICER, (5) SIGITE, (6) ICALT.

Initial articles were selected based on keywords where boolean operators AND and OR were used to refine the searches. More specifically, the search criteria used was: (Improve OR Increase OR Decrease OR Lower OR Reduce) AND (Retention OR Attrition OR Pass OR Fail OR Success) AND (Programming OR Programming Course OR Introductory Programming OR CS1). Applying the criteria resulted in over 1000 somewhat relevant articles being identified, on which two researchers performed an initial inclusion screening. After applying an inclusion screening based on title and abstract, 226 articles remained. From these, further screening was made by including full text content, excluding articles that did not describe a replicable teaching intervention or an intervention that had overall been attempted at least twice, articles that did not discuss CS1 or introductory programming courses, articles that did not include pass-rates before or after the teaching intervention, and articles that did not include the amount of students before or after the teaching intervention (or data from which the numbers could be derived from). Finally, articles that described previously reported results from another perspective and articles from which all results were included in subsequent articles were excluded. The final number of articles used for this study was 32.

2.3 Study Coding

From each article, the following data was extracted:

1. Article details: publication year, name, author(s).
2. Course and institution details.
3. Teaching intervention: year, semester, used practices.
4. Totals and percentages before and after teaching intervention(s): n, pass, fail, withdraw, fail/withdraw.

All three authors performed extraction of details, at times extracting details from the same articles more than once. When conflicts occurred, the first author did an additional review.

From the 32 articles, 60 data entries with information on the amount of students as well as pass-rates before and after the intervention were extracted. When the intervention was done on a separate section during the same semester, the non-intervention and intervention results were paired. If there were multiple sections, an average combining all sections from the same semester was used to reduce possible instructor-related impact on the results. When results were described from subsequent semesters, the pairs were formed so that the pre-pair contained an average of all reported semesters before the intervention, and each post-entry contained details from one reported semester with the intervention. If an article described combined results from multiple institutions, it was included if the totals and percentages before and after the teaching intervention were included, and that the teaching intervention was described in detail.

To utilize a comparable measure of success, the reported totals and percentages in the articles were combined to describe a WDF-rate[1], i.e. the rate of students that did not withdraw, receive a D-grade[2] and did not fail the exam. The WDF-rate is a common measure used to describe course success, and it provides a more realistic view on success due to taking into account students that are not able to continue in their studies (D-grade in some institutions), students that fail in the exam, and students that withdraw from the course before the exam. When the WDF-rate was not readily available and the calculation WDF-grade was not possible, we chose the closest possible number assuming that the metric was the same in both pre- and post-intervention details. In this article, from now on we will use the term pass rate to describe WDF-rate or the closest number available.

We acknowledge that some institutions use a grace period during which students can drop out from the course without any sanctions. Unfortunately, very few articles did report such details, and thus it cannot be taken into account. Similarly, we acknowledge that the grading schema and learning objectives vary among different universities; unless otherwise noted in the articles, our assumption was that the learning objectives and grading schema remained similar between course instances, making the pre- and post-intervention details comparable. Additionally, we did not include demographic details or gender details into the study.

2.4 Classifying Teaching Interventions

Extracting teaching interventions was done in three phases. In the first phase the articles were coded based on the intervention types used. As an example, the article "Combining Cooperative Learning And Peer Instruction In Introductory Computer Science" [4] was coded with tags *cooperative learning*, *student group work*, *team teaching*, and *undergraduate teaching assistants*. The coding reflects the content; the article discusses collaboration and cooperative learning as the main activities, and the peer instruction discussed in the article describes the use of peer instructors, i.e. undergraduate teaching assistants, not to be confused with Peer Instruction by Eric Mazur. To give another example, the article "Experience Report: CS1 for Majors with Media Computation" [19] was coded with tags *media computation* and *peer instruction* as the article mentions that *"...one notable difference between the courses was that the media computation course*

[1]The acronym DFW was also visible in the literature; we utilize the acronym WDF.

[2]Typically 40-49% of the overall course score.

was taught using Peer Instruction in lectures, and the traditional course was not...". Although pair programming is also used, it is not coded as it is used in both non-intervention and intervention groups.

In the second phase, articles were supplemented with additional, descriptive tags. For example, for each article that was coded with the tag *media computation*, tags *content: media*, *contextualization*, *context: media*, were added to describe that the content (material) of the course was updated to contain media-type content, the course was contextualized so that the content had more meaning to the students that participated in the course, and finally, that the context revolved around media. Similarly, for each article that was coded with the tag *peer instruction*, tags *interactive classroom*, *student collaboration*, *reading before class*, *quizzes in class*, and *collaboration in class* were added. No limits were set on the amount of tags that could be used as long as the tags described the intervention properly. For example, an intervention where a game-themed final project was added to the course without additional modifications, the course was not contextualized, but the content was updated to include a game-theme component (tag *content: game-theme*).

Finally, in the third phase, equivalent or closely related tags were combined, and the changes were reflected to the article coding. At the end of the extraction phase, each article had the majority of original tags as well as a set of supplement tags that provided additional information on the described teaching activities. In total, 40 different tags remained after the combination phase, and each of the 60 data entries had on average 3.5 tags.

3. RESULTS

The results of the survey are analyzed from three different viewpoints. First, in Section 3.1 an overview to the results is provided, then in Section 3.2, the most common activities and their effect in the data are discussed, and finally in Section 3.3, the teaching approaches are analyzed based on a primary intervention type.

The results are considered in terms of *realized improvement*, i.e. the absolute improvement divided by the potential, by which the room for improvement that varies between different institutions is taken into account. For example, if a pre-intervention pass rate were 70% and post intervention pass rate were 85%, a potential change of 100-70 = 30% is available for the intervention. Of this, 15/30 or 50% was realized as the absolute improvement, or absolute percentage, was 15%.

3.1 Descriptive Statistics

Table 1 contains descriptive statistics of the data. On average, the pass rates before the intervention were 61.4%, and after intervention 74.4%. Much variance in both pre- and post-passrates exists; the smallest pass rate before intervention was 22.6% and 36% after intervention, while the largest pass rate was 94.2% before intervention, and 92.5% after intervention. The studied student populations varied also a lot. The smallest pre-intervention n was 15 students, which was from a targeted intervention to at-risk students, while the smallest post-intervention n was 9 students; the intervention stragegy in the study was applied to a small summer class. The largest number of students was 2298 before intervention, where the study reported data from past 16 iterations, and 1213 after intervention, which was from a study that reported aggregate results from multiple institutions.

descriptive	min	max	median	mean	sd (σ)
pass rate pre	22.6	94.2	63	61.4	15.5
pass rate post	36	92.5	74	74.4	11.7
students pre	15	2298	148	296.9	487.5
students post	9	1231	86	162.3	200.7

Table 1: Pass rates and study sizes before and after teaching intervention.

Five (8.3%) of the extracted data entries had a negative outcome (the pass rates decreased), while in 91.7% of the entries the intervention had at least a minor improvement on the overall results. On average, before the intervention, there was room for 38.4 percentage units of improvement, while after the intervention there was room for 25.6 percentage units. In other terms, the interventions improved the pass rates on average by 12.8 absolute percentage units, the realized improvement being 33.3% or nearly one third.

3.2 Overall Intervention Effect

Table 2 contains ten most frequent tags and the realized gains in the studies in which they appeared in. While the intervention types cannot be compared with each others due to overlapping, the table provides an overview of the realized improvements over different studies. The intervention tags encompass the following activities:

- *collaboration*: activities that encourage student collaboration either in classrooms or labs
- *content change*: at least parts of the teaching material was changed or updated
- *contextualization*: activities where course content and activities were aligned towards a specific context such as games or media
- *CS0*: the creation of a preliminary course that was to be taken before the introductory programming course; could be organized only for e.g. at-risk students
- *game-theme*: a game-themed component was introduced to the course, e.g. a game-themed project
- *grading schema*: a change in the grading schema; the most common change was to increase the amount of points rewarded from programming activities, while reducing the weight of the course exam
- *group work*: activities with increased group work commitment such as team-based learning and cooperative learning
- *media computation*: activities explicitly declaring the use of media computation (e.g. the book)
- *peer support*: support by peers in form of pairs, groups, hired peer mentors or tutors
- *support*: an umbrella term for all support activities, e.g. increased teacher hours, additional support channels etc.

When considering the median improvement, the studies that had media computation as one of the components were most successful, while studies with a game-theme were the least successful. Facilitating group work and collaboration,

intervention tag	n	min	max	median	avg	σ
collaboration	20	-1	59	39	34	17
content change	36	-17	69	34	34	17
contextualization	17	18	69	37	40	17
CS0	7	18	76	41	43	19
game-theme	9	-39	42	21	18	23
grading schema	11	3	42	30	29	12
group work	7	36	59	44	45	7
media computation	10	24	69	49	48	16
peer support	23	-1	59	36	34	16
support	9	-29	67	36	33	19

Table 2: Ten most common intervention tags and the overall intervention effects of the studies in which they appeared in. Number of studies including the intervention denoted as n, realized pass rates reported using minimum, maximum, median, average and standard deviation (σ) in percentages.

and creating a CS0 course were also among the high-performing activities. While the effect of an intervention activity depends naturally on other activities as well, a noticeable amount of variance was observed even within similar setups. The variance can be explained with the natural variance of student populations over different semesters, student intake, teacher effect, difference in grading criteria among different institutions, and the difference with student workloads among different institutions.

3.3 Primary Intervention Effect

Before comparing the impact of different interventions on programming pass rates, it was first important to determine whether there existed any significant differences in the pre-intervention pass rates of each intervention category, or whether the courses which were included in this study all had a comparable pre-intervention pass rate.

Grouping the 60 pre-intervention pass rates by the five primary intervention categories, a one-way ANOVA was performed. A Shapiro Wilk test confirmed the pass rates were normally distributed for all groups ($p > .05$), with the exceptions of relatable content and contextualization ($p = .01$), and hybrid approaches ($p = .01$). However as violations from normality do not substantially affect the type I error rate, and an ANOVA is considered relatively robust against this violation, we proceeded. Homogeneity of variances was confirmed by Levene's test ($p = .298$). A one-way ANOVA showed that there were no statistically significant differences in pre-intervention pass rates for the five primary intervention categories used in this study, $F(4,\ 55) = 2.17$, $p = .084$. To ensure that the violation of normality had not impacted the test result, we also performed a Kruskal-Wallis test, which confirmed that there were no statistically significant differences between the pre-intervention pass rates of each group, $\chi^2(4) = 9.13$, $p = .18$.

3.3.1 *Collaboration and Peer Support*

Approaches that include collaboration and peer support include peer-led team learning activities [9], pair programming activities [28] and cooperative and collaborative practices [4, 24]. Results are shown in Table 3. A total of 14 studies were classified as having applied an intervention, which primarily consisted of moving towards a collaborative, or peer support based approach. Three specific approaches were identified: cooperative learning (3 courses), team based learning (5 courses), and pair programming (6 courses). Out of all the interventions that were explored in this study, cooperative learning was found to yield the largest absolute improvement in CS1 pass rates (25.7% on average), and team based learning was found to yield the second largest absolute improvement (18.1% on average). Despite being frequently cited as an enabler for programming skills, the pair programming approach was only found to yield a absolute improvement of 9.6% on average, and ranked 11 out of the 13 interventions that were explored by this study. It was possible that courses to which this intervention was applied already had good pass rates, and therefore there was little scope for absolute improvement. When considering realized changes, we note that on average, pair programming yielded a realized increase of 27% in pass rates, but overall, this approach was still ranked 11th out of the 13 interventions which were explored by this study. Considering the results of all 14 courses combined, we found that instructors who applied a collaborative or peer support based intervention generally received the largest improvements in pass rates when compared to the other groups examined in this study (16.1% improvement, realized change 34.3%). A possible explanation is that the continuous feedback and cooperation from peers acts as an enabler for programming skills, supplementing feedback received from the compiler, which is not always at a sufficient level for inexperienced students to comprehend.

3.3.2 *Bootstrapping*

Bootstrapping practices either organized a course before the start of the introductory programming course [20, 7] or started the introductory programming segment using a visual programming environment such as Scratch and Alice [11]. Some of the activities were also targeted at at-risk students [16]. Results are shown in Table 4. A total of 9 studies were classified as having applied such an intervention. Two specific approaches were identified: using visual programming tools such as Scratch or Alice (5 courses), and introducing CS0 (4 courses). Out of all the interventions that were explored in this study, using visual programming tools were found to yield the fifth largest absolute improvement in pass rates (17.3% on average). A similar high ranking was found when considering realized improvement (fourth, 38.6%), which positioned using visual programming tools as the fourth overall best intervention. Whilst the absolute improvement for courses that introduced CS0 was much lower than visual programming (10.5% increase), the realized change that was yielded by this intervention was comparable (34.9% increase). Considering the results of all 9 courses combined, we found that instructors who applied a bootstrapping intervention generally received the second largest improvements in pass rates when compared to the other groups examined in this study (absolute change 14.3%, realized change 37.0%). It is possible that the initial simplification offered by these forms of intervention are able to assist students who might otherwise fail CS1, by suppressing the syntax barrier until they have gained sufficient knowledge of the underlying concepts. This also ties into research on threshold concepts, which suggested that reducing the level of complexity initially may be an effective way to assist students in overcoming thresholds.

3.3.3 *Relatable Content and Contextualization*

Approaches that introduced relateable content sought to make programming more understandable to students. These approaches include Media Computation [21], introducing real world projects [5] as well as courses that evolve around games [1]. Results are shown in Table 5. A total of 14 studies were classified as having applied an intervention, which primarily consisted of using relatable content and contextualization as a means to improve CS1 pass rates. Two specific approaches were identified: media computation (7 courses), and gamification (7 courses). Out of all the interventions that were explored in this study, using media computation was found to yield the seventh largest absolute improvement in pass rates (14.7% on average), and a comparable improvement was found for gamification (10.8% on average). However, when considering realized changes, media computation was found to yield the largest realized change across all interventions explored in this study (50.1% increase), whereas gamification was found to only yield the tenth largest (27.4% increase). Overall, and considering the results of all 14 courses combined, we found that instructors who applied a relateable content or contextualization intervention generally received the third largest improvements in pass rates when compared to the other groups examined in this study (absolute change 11.6%, realized change 38.7%). As media computation (overall rank 2) considerably outperformed gamification (overall rank 10), it could be the case that whilst games provide a useful tool to contextualize a learning task, there are still fundamental underlying programming concepts that can be better served by adopting a media computation approach.

3.3.4 *Course Setup, Assessment, Resourcing*

Approaches that modify course setup, assessment and resourcing included a broad range of practises starting from adjusting course content based on data from an assessment system [18], introducing new content, a programming tool that provides additional support and changing the grading schema assessment [15, 12]. Results are shown in Table 6. A total of 15 studies were classified as having applied an intervention which primarily consisted of changing aspects of the course setup, rather than changing elements of the teaching approach. Three specific approaches were identified: changing class size (4 courses), improving existing resources (2 courses), and changing assessment criteria (9 courses). Overall, the largest absolute improvements in pass rates were found by changing the class size (17.8% improvement) and improving existing resources (17.5%). However, when considering these improvements relatively, they were among the five worst interventions found by this study. Similarly, making changes to the assessment criteria applied in the course yielded on average an absolute improvement of 10.1% and realized improvement of 22.5%. But when considering these changes against the other 13 interventions explored by this study, changing assessment criteria ranked 12th. Considering the results from all 15 courses combined, we found that instructors who applied an intervention based on course setup generally yielded the fourth largest improvements in pass rates when compared to the other groups (absolute change 13.4%, realized change 26.8%). The findings on changing class size to improve pass rates are consistent with previous studies [3] that have suggested that smaller classes generally have lower failure rates than larger ones. However, overall, it is possible that this group of interventions were ranked as one of the lowest because making changes to the course setup, such as the assessment criteria, do nothing to adjust the likelihood of a student overcoming thresholds understanding programming concepts.

3.3.5 *Hybrid Approaches*

Hybrid approaches are approaches that upon discussion were not included in any of the primary categories. These include combinations of different practices [19, 14, 8]. Results are shown in Table 7. A total of 8 studies were classified as having applied an intervention, which primarily consisted of combining several different teaching interventions to yield a hybrid approach. Three combinations were identified: media computation with pair programming (2 studies), extreme apprenticeship (3 courses), and collaborative learning with relateable content (e.g. games) (3 courses). Overall, combining media computation with pair programming, or adopting an extreme apprenticeship approach were found to yield mid-range improvements in pass rates, ranging from 13.5-16.5% in absolute terms, or 36.9-49.3% in realized terms. These approaches were ranked fifth and seventh among the overall 13 interventions that were explored in this study. However, combining collaborative learning with content was found to be the worst overall intervention, actually yielding a decrease in pass rates of 9.7%, or 53.7% in realized terms. However, we note that some of the courses, which switched to this approach already had a very high pass rate ($> 90\%$), and therefore the scope for improvement was minimal.

3.3.6 *Comparing Primary Interventions*

The final question, which remained from this study, was to determine whether there were any significant differences in the post-pass rates of studies that applied different types of interventions. Grouping the 60 post-intervention pass rates by the five primary intervention categories, a one-way ANOVA was performed. A Shapiro Wilk test confirmed the pass rates were normally distributed for all intervention groups ($p > .05$) and homogeneity of variances was confirmed by Levene's test ($p = .487$). A one-way ANOVA showed no statistically significant differences in the post-intervention pass rates for the five primary intervention groups of this study, $F(4,\ 55) = 2.02$, $p = .105$. Similarly, a Tukey post-hoc analysis revealed no significant pairwise differences in post-intervention pass rates. This suggests that whilst substantial improvements in pass rates can be achieved by applying different interventions, the overall pass rates after applying different types of intervention are not substantially different.

4. DISCUSSION

The interventions reported in the literature increase introductory programming course pass rates by one third on average. A large part of the reported interventions increase student and teacher collaboration and update the teaching material and content in an attempt to make the content more relatable to the students. Support is facilitated in many ways; one approach is recruiting peer tutors that help students as they are working, while another approach is to build a CS0-course which acts as a bridge to the programming studies. Some interventions also changed the grading schema, which is known to affect students' behaviour. What may be missing however, are the reports on interventions

		Absolute Change			Realized Change			
Intervention	Courses	Mean	SD	Rank	Mean	SD	Rank	Overall
Cooperative	3	25.7	3.8	1 / 13	47.7	10.0	3 / 13	1 / 13
Team Based	5	18.1	11.6	2 / 13	35.0	12.3	6 / 13	3 / 13
Pair Programming	6	9.6	10.1	12 / 13	27.0	23.7	11 / 13	11 / 13
Overall Intervention	14	16.1	16.1	1 / 5	34.3	18.6	3 / 5	1 / 5

Table 3: Improvements in Pass Rates for Courses which applied Collaborative and Peer Support Interventions

		Absolute Change			Realized Change			
Intervention	Courses	Mean	SD	Rank	Mean	SD	Rank	Overall
Scratch and Alice	5	17.3	18.7	5 / 13	38.6	30.8	4 / 13	4 / 13
CS0	4	10.5	4.4	10 / 13	34.9	9.5	7 / 13	9 / 13
Overall Intervention	9	14.3	12.9	2 / 5	37.0	20.3	2 / 5	2 / 5

Table 4: Improvements in Pass Rates for Courses which applied Bootstrapping

		Absolute Change			Realized Change			
Intervention	Courses	Mean	SD	Rank	Mean	SD	Rank	Overall
Media Computation	7	14.7	5.4	7 / 13	50.1	18.9	1 / 13	2 / 13
Games	7	10.8	6.0	9 / 13	27.4	8.3	10 / 13	10 / 13
Overall Intervention	14	12.7	11.6	4 / 5	38.7	18.3	1 / 5	3 / 5

Table 5: Improvements in Pass Rates for Courses which applied Relatable Content and Contextualization

		Absolute Change			Realized Change			
Intervention	Courses	Mean	SD	Rank	Mean	SD	Rank	Overall
Class Size	4	17.8	16.6	3 / 13	34.0	43.2	8 / 13	6 / 13
Resource Improvement	2	17.5	2.8	4 / 13	32.1	5.2	9 / 13	8 / 13
Assessment	9	10.5	9.9	11 / 13	22.5	19.4	12 / 13	12 / 13
Overall Intervention	15	13.4	18.8	3 / 5	26.8	27.4	4 / 5	4 / 5

Table 6: Improvements in Pass Rates for Courses which applied Course Setup Interventions

		Absolute Change			Realized Change			
Intervention	Courses	Mean	SD	Rank	Mean	SD	Rank	Overall
Media Computation with Pair Programming	2	13.5	5.4	8 / 13	49.3	0.2	2 / 13	5 / 13
Extreme Apprenticeship	3	16.5	1.9	6 / 13	36.9	4.2	5 / 13	7 / 13
Collaboration with Games	3	-9.7	12.2	13 / 13	-53.7	67.9	13 / 13	13 / 13
Overall Intervention	8	6.0	13.2	5 / 5	6.0	61.6	5 / 5	5 / 5

Table 7: Improvements in Pass Rates for Courses which applied Hybrid Learning Approaches

that did not yield an improvement. Thus, educators that have tried an intervention but received poor results should also be encouraged and supported in reporting the results to create a more stable picture of the field.

Whilst no statistically significant differences between the effectiveness of the teaching interventions were found, marginal differences between approaches exist. The courses with relatable content (e.g. using *media computation*) with cooperative elements (e.g. pair programming) were among the top performers with CS0-courses, while courses with pair programming as the only intervention type and courses with game-theme performed more poorly when compared to others. However, these interventions were still able to improve pass rates by a minimum of 10%, suggesting that although they were not as strong as the other interventions in this study, they were still beneficial when compared to the traditional lecture and lab approach they replaced. Do note however, that many of the interventions did combine practices together, and e.g. the effect of a possible change in teaching material may be left unreported.

Nevertheless, the data confirms that educators and researchers are making a difference when trying out new teaching interventions and pedagogical approaches. One of the common denominator among all the interventions is change, while the other side of the coin – the "past situation" – may often be a state of complacency.

4.1 Related Work

While no such review exists from the field of introductory programming, many reviews on the influences of teaching approaches exist from other fields. The most notable one, a synthesis of 800 meta-analyses on teaching and achievement in schools by John Hattie [6], combines the results of multiple meta-analyses to form a picture of the efficiency of different teaching approaches. Although the results from Hattie's study are from schools, they provide an additional viewpoint to our findings. As an example, while collaborative and cooperative approaches were among the most effective approaches in this study, in schools the practices that require cooperation are above average in efficiency but increase in efficiency as students get older [6]. Similarly, peer tutoring in schools is not as efficient as in our study, which further provides support on the suggestion that the ability to support others and work in teams increases with age.

While many of the effects observed by Hattie were related to the teacher, such as the teacher clarity and instructional quality, little focus was on these aspects. The closest matches from our tagging are different feedback approaches and improving the course content; the first is also a part of the collaborative approaches.

4.2 Limitations

As the results presented in this article are derived from a synthesis of the results from other articles, a number of validity concerns, including the justification of the synthesis approach, can be raised which are discussed in this section.

Firstly, the teaching approaches that were used prior to the intervention were rarely explicitly stated in a very detailed fashion. For the most part, it was implied from the articles that the current approach was one based upon a traditional lecture and lab based approach. It is possible that in some cases important details were not reported in the articles, and thus were missed in the encoding process. Similarly, the learning objectives of the introductory programming courses were not considered for this study.

Secondly, the soundness of the teaching intervention design and experiments were not judged, and e.g. the quality of the used teaching material was not considered. It is likely that some interventions were implemented and executed better than others, which may lead to an imbalance in the results. Thus, one should not refrain from trying out the approaches that didn't seem to work.

Thirdly, the final number of selected articles, $n = 32$ is low, especially when considering that introductory programming courses have been studied for decades. A major concern in our case is the possibility of selective reporting. As only 8.3% of the studies contained negative results, it is possible that interventions with negative results have not often been reported. To counter this, we explored sources of *gray literature* via generalized searches, but our efforts were largely unsuccessful. The tendency to only encounter results for studies where an intervention has been successful however is a common limitation of systematic reviews.

Fourthly, whilst the results suggest that almost any planned intervention improves the existing state, the results have been gathered from studies that study university-level introductory programming courses. Further analysis should be performed when applying the results in other contexts to verify whether such approaches can yield similar improvements in pass rates at all levels of education.

Fifthly, the possible teacher effect and the effect of different student populations among different institutions is not taken into account. This is a deliberate choice due to the small amount of final articles.

Sixth, the choice for primary interventions and the tagging methodology has an inherent limitation as many of the studies had more than one intervention. Thus, the results shown here depend on the used classification, and different classification approaches may yield different results.

Finally, we note the unavoidable limitation that the assessment criteria of the individual courses were not the same over all data entries. Studies within the UK generally defined 'pass rate' as consisting of those students who had scored over 40% in the course. However, other studies defined 'pass rate' as consisting of those students who had scored at least a 'C', and others defined 'pass rate' as consisting of those students who had scored anything apart from an 'F'. Other studies did not supply details at all. Therefore this study unavoidably has to assume that a consistent notion of 'pass rate' exists and holds valid across the different teaching contexts. However, we note that this is a common limitation of other studies of this nature (e.g. [3, 25]).

5. CONCLUSION

In this article, we performed a quantitative systematic review on articles describing introductory programming teaching approaches, and provided an analysis of the effect that various interventions can have on the pass rates of introductory programming courses. While the total amount of articles was relatively low, a total of 60 pre-intervention and post-intervention pass rates, describing thirteen different teaching approaches, were extracted and analyzed.

The results showed that on average, teaching interventions can improve programming pass rates by nearly one third when compared to a traditional lecture and lab based approach. While no statistically significant differences between the effectiveness of teaching interventions were observed, marginal differences do exist. The courses with relatable content (e.g. using *media computation*) with cooperative elements (e.g. pair programming) were among the top performers with CS0-courses, while courses with pair programming as the only intervention type and courses with game-theme performed more poorly when compared to others.

What the results of this analysis mean in practice, is that educators and researchers that are applying teaching interventions are making a difference. Whilst there is no silver bullet, no teaching approach works significantly better than others, a conscious change almost always results in an improvement in pass rates over the existing situation.

6. REFERENCES

[1] J. D. Bayliss. The effects of games in CS1-3. In *Microsoft Academic Days Conference on Game Development in Computer Science Education*, pages 59–63. Citeseer, 2007.

[2] M. Ben-Ari. Constructivism in computer science education. In *SIGCSE bulletin*, volume 30, pages 257–261. ACM, 1998.

[3] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *SIGCSE Bulletin*, 39(2):32–36, 2007.

[4] J. D. Chase and E. G. Okie. Combining cooperative learning and peer instruction in introductory computer science. *SIGCSE Bulletin*, 32(1):372–376, Mar. 2000.

[5] N. De La Mora and C. F. Reilly. The impact of real-world topic labs on student performance in CS1. In *Proc. Frontiers in Education*, pages 1–6. IEEE, 2012.

[6] J. Hattie. *Visible learning: A synthesis of over 800 meta-analyses relating to achievement*. Routledge, 2013.

[7] M. Haungs, C. Clark, J. Clements, and D. Janzen. Improving first-year success and retention through interest-based CS0 courses. In *Proc. SIGCSE*, pages 589–594. ACM, 2012.

[8] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proc. SIGITE*, pages 3–8. ACM, 2011.

[9] P. Lasserre and C. Szostak. Effects of team-based learning on a cs1 course. In *Proc. ITiCSE*, pages 133–137. ACM, 2011.

[10] I. Milne and G. Rowe. Difficulties in learning and teaching programming - views of students and tutors. *Educ. and Information Technologies*, 7(1):55–66, 2002.

[11] P. Mullins, D. Whitfield, and M. Conlon. Using alice 2.0 as a first language. *Journal of Computing Sciences in Colleges*, 24(3):136–143, 2009.

[12] U. Nikula, O. Gotel, and J. Kasurinen. A motivation guided holistic rehabilitation of the first programming course. *Trans. Comput. Educ.*, 11(4):24:1–24:38, Nov. 2011.

[13] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *SIGCSE Bulletin*, volume 39, pages 204–223. ACM, 2007.

[14] L. Porter and B. Simon. Retaining nearly one-third more majors with a trio of instructional best practices in CS1. In *Proc. SIGCSE*, pages 165–170. ACM, 2013.

[15] D. Radošević, T. Orehovački, and A. Lovrenčić. New approaches and tools in teaching programming. In *Proc. of Central European Conference on Information and Intelligent Systems*, pages 49–57, 2009.

[16] M. Rizvi and T. Humphries. A scratch-based cs0 course for at-risk computer science majors. In *Proc. Frontiers in Education*, pages 1–5. IEEE, 2012.

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

[18] S. C. Shaffer and M. B. Rosson. Increasing student success by modifying course delivery based on student submission data. *ACM Inroads*, 4(4):81–86, Dec. 2013.

[19] B. Simon, P. Kinnunen, L. Porter, and D. Zazkis. Experience report: CS1 for majors with media computation. In *Proc. ITiCSE*, pages 214–218. ACM, 2010.

[20] R. H. Sloan and P. Troy. CS 0.5: A better approach to introductory computer science for majors. *SIGCSE Bulletin*, 40(1):271–275, Mar. 2008.

[21] A. E. Tew, C. Fowler, and M. Guzdial. Tracking an innovation in introductory CS education from a research university to a two-year college. In *Proc. SIGCSE*, pages 416–420. ACM, 2005.

[22] I. Utting, A. E. Tew, M. McCracken, L. Thomas, D. Bouvier, R. Frye, J. Paterson, M. Caspersen, Y. B.-D. Kolikant, J. Sorva, and T. Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proc. ITiCSE Working Group Reports*, pages 15–32. ACM, 2013.

[23] A. Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Proc. ICALT*, pages 498–499. IEEE, 2013.

[24] H. M. Walker. Collaborative learning: a case study for CS1 at grinnell college and austin. In *SIGCSE Bulletin*, volume 29, pages 209–213. ACM, 1997.

[25] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *To appear in Proc. Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 2014.

[26] C. Watson, F. W. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proc. ICALT*, pages 319–323. IEEE, 2013.

[27] C. Watson, F. W. Li, and J. L. Godwin. No tests required: comparing traditional and dynamic predictors of programming success. In *Proc. SIGCSE*, pages 469–474. ACM, 2014.

[28] L. Williams, C. McDowell, N. Nagappan, J. Fernald, and L. Werner. Building pair programming knowledge through a family of experiments. In *Proc. Empirical Software Engineering*, pages 143–152. IEEE.

APPENDIX

Due to space limitation and to serve as a starting point for future researchers, a list of 32 references is provided at http://bit.ly/1qkb8GI.

Publication I.2

Arto Vihavainen, Matti Paksula, and Matti Luukkainen

Extreme Apprenticeship Method in Teaching Programming for Beginners

In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE '11)*

Extreme Apprenticeship Method in Teaching Programming for Beginners

Arto Vihavainen, Matti Paksula and Matti Luukkainen
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ avihavai, paksula, mluukkai }@cs.helsinki.fi

ABSTRACT

Learning a craft like programming is efficient when novices learn from people who already master the craft. In this paper we define *Extreme Apprenticeship*, an extension to the cognitive apprenticeship model. Our model is based on a set of values and practices that emphasize *learning by doing* together with *continuous feedback* as the most efficient means for learning. We show how the method was applied to a CS I programming course. Application of the method resulted in a significant decrease in the dropout rates in comparison with the previous traditionally conducted course instances.

Categories and Subject Descriptors

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

General Terms

Design, Human Factors

Keywords

cognitive apprenticeship, course material, continuous feedback, instructional design, programming education, motivation, best practices, learning by doing

1. INTRODUCTION

Teaching programming is hard. Lots of research from many different perspectives has been devoted to the topic during the past couple of decades (see eg. [23, 21]), but there is still no consensus on what is the most effective way to teach programming. Most universities are still using a traditional format in the introductory programming courses (CS I courses). The traditional format consists of lectures, take-home assignments and perhaps also demo sessions where model solutions to the exercises are shown (see eg. [7, 24]). Lectures tend to be structured according to the language constructs, rather than the more general application strategies. This approach is used despite various research results [31, 23, 25] indicating that the problem is not to learn the syntax or semantics of individual language constructs, but to master the process on how to combine constructs to meaningful programs.

The language constructs introduced in lectures are typically applied in programming exercises. With very little support to the programming process, doing exercises is hard for part of the student population [7, 24], to those who in the literature are characterized as stoppers [22] or ineffective novices [23]. Many of these end up dropping the course due to not being able to solve problems and therefore feeling inadequate. Another problem of take-home exercises is that students may learn bad work habits from solving the problems by themselves.

The context in which students do exercises themselves can be regarded as a minimally guided environment. It is well known in educational psychology (see e. g. [20]) that, due to the nature of human cognitive architecture, a minimally guided approach is not optimal for novices learning a cognitively challenging task, such as programming.

In this paper we will describe a variation of *Cognitive Apprenticeship* called *Extreme Apprenticeship* that has a strong emphasis on guided programming exercises. We also report the experiences from its first application at the University of Helsinki Department of Computer Science.

2. PEDAGOGICAL BACKGROUND

The dropout rates of introductory programming courses tend to be high[1], so it is quite evident that the traditional approach shoud be improved.

One of the most interesting approaches in programming instruction is Cognitive Apprenticeship Model [9, 10], where the focus is on the process rather than just on the end products. Cognitive Apprenticeship also puts a heavy emphasis on optimizing coaching and guidance available to the students.

Numerous studies have shown that both the motivation and the comfort level of students have a remarkable effect on learning [5]. Cognitive Apprenticeship already has many ingredients to boost both, but also the role of programming exercises is remarkable.

[1]E.g. in University of Helsinki the long term average dropout rate has been c.a. 45 %.

2.1 Cognitive Apprenticeship

The Cognitive Apprenticeship Model has recently had many applications in teaching programming with positive results (see e. g. [1, 6, 8, 16]). The model is based on the ancient model of apprenticeship education where a profession is learned while working under the guidance of a senior master. Traditionally the apprenticeship model has been mostly applied in context of learning professions that require physical skills such as shoe making. In Cognitive Apprenticeship the emphasis is more on acquiring cognitive skills.

The key observation in Cognitive Apprenticeship is that when teaching novices, masters of a skill do not usually take into account the complex process that leads to end products [9, 10]. As stated previously this is far too common also in teaching programming.

Cognitive Apprenticeship divides instruction into three stages: modeling, scaffolding and fading. In the *modeling* stage the teacher gives students a conceptual model of the process, with which an expert performs the task under study. One effective way of modeling is to base the lectures on *worked examples* [8] instead of concentrating on language structures. A worked example shows e.g. completion of a programming task from start to finish. While completing the task, the teacher is thinking aloud all the time, explaining the decisions made during the process.

After the modeling stage, students move to the *scaffolding* stage. Typically this means that the students are exposed to exercises that are made under the guidance of an experienced instructor. Scaffolding refers to the way support is given to the students. The key idea is that students are not given straight answers, but rather just enough hints to be able to discover the answers to their questions themselves. Scaffolding is based on Vygotsky's idea that learning is most efficient when a student is given just enough information that is enough to boost the student's ability to finish the task [27].

When the student starts to master a task by himself, the scaffolding should be dismantled. This is the *fading* stage of apprenticeship learning.

The Apprenticeship-based approach to learning programming seems to be advocated also by the Agile and Software Craftsmanship people in the industry, such as Robert Martin who has stated that *"Software is a craft that takes years to learn, and more years to master. The only way to properly learn the craft is to be taught at the side of a master"* [19]. Martin calls for apprenticeship-type mentoring to the software industry, where the recently graduated apprentices would work in a software project context with constant interactive guidance by journeymen and masters.

2.2 The roles of programming exercises

Somewhat surprisingly the applications of Cognitive Apprenticeship to programming instruction have not had much emphasis on the role of programming assignments. It seems evident that the exercises are crucial in learning programming, and there also exists empirical data to support this fact [12].

The *Active Learning* [14]-based methods (eg. [28, 11]) do raise the programming activity of students to a big role, but seem to still stress collaborative aspects more than individual effort.

Programming exercises can have an even more important role than just applying the theory taught in lectures, as Roumani [24] stated *"we think of them (assignments) as teaching instruments that complement lectures by teaching the same material but in an exploratory fashion"*.

In addition to being an important learning instrument, programming exercises have a huge impact on the motivation of the students. It is well known that the level of motivation correlates positively to success in learning [15, 18]. Empirical evidence for this exists also from the field of programming instruction [5].

It has especially been shown that students who are performing activities for the activities themselves, i.e., intrinsically motivated students perform better than those who seek extrinsic rewards [17]. Giving too difficult programming assignments is a certain way to kill the motivation of weaker students, but suitably challenging and relevant exercises with short-term goals that students can achieve are known to raise intrinsic motivation [17, 26, 18].

The way students get instructional feedback also has an effect on their motivation. Talking with students about their solutions and problem solving strategies while giving them hints on how to improve them is known to have a positive impact on student motivation [18], so from the motivation point of view, the type of programming exercises and the guidance available when solving exercises are crucial for the effectiveness of the scaffolding phase in the apprenticeship type of instruction.

Besides motivation, the *comfort level* of a student has been shown to have a remarkable impact on learning (see eg. [5, 30, 29]. Their comfort level incorporates students orientation to themselves (self-esteem) and judgement of their capabilities to execute the required tasks (self-efficiency) [2, 5]. According to Bandura [2], the most important source of self-efficiency is the student's evaluation of the outcomes of his attempts to perform activities. Thus, suitable exercises with proper guidance and feedback are an essential tool for building students' comfort levels.

3. EXTREME APPRENTICESHIP METHOD

One of the ideas in *Extreme Programming* [4] is to take a group of software development best practices and take those to the extreme levels. For example, in order to improve the quality of written code, development teams should have code reviews. In Extreme Programming this practice becomes integrated as a technique called pair programming where the practice is taken to an extreme level: code is written under constant reviewing.

We took a similar approach in teaching of programming where we constructed our method on top of the Cognitive Apprenticeship model. Especially the scaffolding stage of the model is stressed.

Extreme Apprenticeship Method

The following values are stressed during all the course activities:

- **Learning by doing**. The craft will only be mastered by actually practicing it.
- **Continuous feedback**. Continous feedback must be implemented in both directions. The student receives multi-level feedback from his progress and instructors, and the instructor receives feedback by monitoring the students progress and challenges.

- **No compromise**. The skills to be learned are practiced as long as it takes for each individual.

- **An apprentice becomes a master**. The ultimate goal of instruction should be that the student will eventually become the master.

The values above induce a set of the following practices that are applied in actual course implementation:

- **Avoiding tons of preaching**. Since the effectiveness of lectures in teaching programming is questionable, the lecturing should cover only the bare minimum to get started with exercises.
- **Relevant examples**. Topics covered in the lectures have to be relevant for the exercises.
- **Start early**. Exercises start right after the first lecture of the course. During the first weeks of the course all the students are already solving an extensive amount of simple exercises. This gives all the students a strong routine of code writing and a motivation boost right at the start of the course.
- **Help available**. Exercises are completed in a lab in the presence of instructors who are offering the scaffolding style of guidance.
- **Small goals**. Exercises are split into small parts with clearly set intermediate goals.

  These small intermediate steps guarantee that students feel that they are learning and making progress all the time.
- **Exercises are mandatory**. Since the exercises are the main instrument in learning, the majority of the exercises are mandatory for all the students.
- **Train the routine**. The amount of exercises should be high and to some extent repetitive in their nature.
- **Clean guidelines**. Exercises have to provide clear starting points and structures on how to start solving the task.
- **Encourage to look for information**. While doing the exercises students are also required to find out things that are not covered during the lectures.

4. APPLYING THE METHOD

The method was applied in introductory programming courses at the Department of Computer Science at the University of Helsinki. For administrative purposes the one semester CS I introductory Java programming course is given in two separate parts. The courses *Introduction to Programming* and *Advanced Programming* are taught as separate units where Advanced programming further deepens the knowledge built during the Introduction to programming course. Both parts last 6 weeks, totalling the length of one semester.

Introduction to programming covers assignment, expressions, terminal input and output, basic control structures, classes, objects, methods, arrays and strings. Advanced programming concentrates on advanced object oriented features such as inheritance, interfaces and polymorphism, and familiarizes students with the most essential features of Java API, exceptions and file I/O.

4.1 Study material and lectures

The study material and lectures play a key role in the modeling phase in teaching the skills to be learned. On the other hand, as programming is a craft, it requires plenty of practice.

In order to avoid tons of preaching we reduced the number of lectures from the usual 5 hours per week to just 2 hours. Lectures and the supporting material did not even try to cover every detail of the language. Rather only the required overview for the exercises was given and students were supported and encouraged to look for information themselves.

All the material shown in the lectures was available to students on-line. The material was a web page, written in book-like format. The material followed the structure of exercises, allowing students to read the material as they proceeded with the exercises, providing scaffolding for the actual process of learning by doing.

In the material and lectures all the constructs were always presented with relevant examples from the point of view of exercise solving. This allowed students to remember that the programming tasks in exercises were often just variations of the examples shown in the lectures.

In addition to knowing a collection of language constructs, problem-solving skills are needed in programming. In the material and the lectures the main idea was to give worked examples, not just to show working code or show direct answers, but to demonstrate step by step how a solution could be devised for a problem. This approach helped students to identify good ways of solving programming problems already during the lectures.

4.2 Exercises

It is expected that students use most of the time they devote to the course in active solving of programming exercises. This trains the routine and gives a constant feeling of success by achieving small goals. The exercises especially in the beginning of the course were aimed to build up programming routines and confidence, partly motivated by the Software Craftsmanship community's idea of *Code Katas*, which are small exercises which help programmers to improve their skills through practice and repetition. As Corey Heines puts it *"practising the solution to a Kata until the steps and keystrokes became like second nature, and you could do them without thinking. In this way, you can internalize the process/technique you are practicing until it is under your fingers"* [13].

For each week we introduced a set of new exercises, an amount ranging from 15 to almost 40. Most of the initial exercises were small, like "output numbers from 1 to 99". Sequentially done small exercises combined as bigger programs. This approach in composing bigger programs showed students how to split a big task to smaller sub-tasks – a vital skill in programming.

The exercise difficulty was worked out to be incremental. The first ones of the weekly exercises were used to "warm up" students, providing the first small goals to get started and keeping students in their comfort level.

Each task had a short textual description of the expected behavior of the program. Two additional implementations of technical scaffolding were also introduced: Output- and Main-driven Programming. These two techniques provided additional support for the student.

Output-driven Programming

Similarly to *Test-driven Development* [3] where the unit test for the code is implemented first, our exercises showed the output of the program that the student was supposed to match with his implementation. A typical exercise looked like this:

```
"Write a program that asks user's name and then
 outputs it"

Give your name: Matti

Hello, Matti!
```

This allowed students to understand the textual description of the task better. The expected output also allowed students to verify that their program is working correctly and the small goal is achieved.

The expected output can also provide additional hints for structuring the program. An example of this is shown in the next example.

```
"Write a program that reads a number from the user.
 The program checks if the range of the number is
 between 0 and 100."

Give a number: -2
Please give a number between 0 and 100!

Give a number: 102
Please give a number between 0 and 100!

Give a number: 2
Thank you!
```

From the above output it is possible to determine required parts and their behaviors, providing a starting point for the implementation: the output suggests that there is some kind of loop in the program code combined with reading and conditions.

Main-driven Programming

Later when the tasks became more complicated *Main-driven Programming*, an extension of Output-driven Programming, was introduced. We gave a small testing program that could be inserted into the main method of a Java program.

In the next example the task is to design a `TravelCard`-class, which would have an owner and balance.

```
Copy this to your main-method:

  TravelCard artosCard = new TravelCard("Arto");
  System.out.println(artosCard);

Expected output:

  Owner Arto, balance 0.0 euros
```

To complete this task the student has to create a new class named `TravelCard` and figure out how to implement a `toString()`-method and required attributes for the class. This ensures on some level that the structure in the final program will be good.

4.3 Exercise Sessions

Exercise sessions were organized in computer labs where students worked to solve the exercises. Help was continuously available during the exercise sessions in the form of teachers and teaching assistants, e.g. the instructors. Anyone could enter the class without having to reserve a specific slot. Every week had 8 hours of exercise sessions, and students could attend as many sessions as needed.

An important principle in our approach was that the programming started as early as possible. The first exercise session was right after the starting lecture of the course. For the first week the students already had 30 small exercises. Due to the guidance available in exercise sessions even those with no previous experience of programming managed well with the start early approach: 88% of the students finished over 25 exercises during the first week. The quick and encouraging start raised the self-confidence and comfort level of students, and also had an immense effect on their motivation.

In order to enforce good programming habits, students had to have their finished solutions accepted by the instructors. If an instructor noticed a flaw in the approach (bad naming or indentation, too complex solution logic for the problem, etc.), he pointed it out, and the student had to redo parts of the exercise. In general we allowed no compromises in the solutions of students. This way, each student refined their solutions to the point where the solutions could be passed as ”model answers”.

4.4 Continuous Feedback

During the course we implemented continuous feedback to provide fast evaluation and a continuous feeling of progress for the students. During the exercise sessions students received positive reinforcement in the form of instructors that were aiding them forward.

If a student did not have specific questions during the exercise session, the instructors still actively engaged with him to make sure he was working towards the right direction. If something to correct was noticed, the instructor nudged the student to the right direction by questioning the approach or by providing constructive feedback. This was the key continuous feedback as the hints received during the learning process are essential for acquiring good programming and problem-solving habits. Instructors were not allowed to give direct solutions to the exercises, and the key idea was to support the students so that they could figure out the solutions themselves.

In addition to instructor feedback, students had their completed exercises marked down to a check-list, allowing them to see the check-list filling with marked exercises. We feel that the list played an important role in feedback; every check was a small victory. Check-lists were also updated to the course web-page at the end of every day, allowing students to see the progress of other students as well.

In addition to evaluation during exercises, students were evaluated with 3 small biweekly exams done with the computer and a final traditional exam. Small exams provided valuable feedback for students and also to instructors throughout the courses.

The final exam was constructed to be as similar as possible to the usual programming exams conducted at our university to provide meaningful comparison of the course results. The exam was a paper exam consisting mostly of programming on paper. It was not allowed to use any material in the exam. A student had to get 50 % of the total maximum score in order to pass the course.

5. COURSE RESULTS

The introductory programming courses at the Department of Computer Science at the University of Helsinki are taught during both fall and spring semesters. Fall semesters consist mostly of students who are majoring in computer science, while spring semesters have mostly students who have computer science as a minor subject. Some of the students minoring in computer science participate only in the Introduction to programming course and do not proceed to Advanced programming.

Until spring 2010 the introductory programming courses have followed the traditional lecture and take-home exercise model. The first course implementation following Extreme Apprenticeship was during the spring semester 2010.

Next we will compare the outcome of the Extreme Apprenticeship-based course to the previous course instances from past 8 years in terms of percentage of passed students. The results are reported separately in the tables below for Introduction to programming and Advanced programming. The Extreme Apprenticeship-based implementation in spring 2010 is highlighted using bold face. The column titled n denotes the number of total participants.

As stated in the previous section, the paper exam in the spring 2010 implementation was similar to the ones that had been used in the course for years already. Because it has always been a requirement to get 50% of the exam score to pass the course, the numbers should be comparable for all the course implementations.

Introduction to Programming

	n	passed
s02	92	38.0 %
f02	332	53.6 %
s03	98	39.8 %
f03	261	64.0 %
s04	84	61.9 %
f04	211	59.2 %
s05	112	46.4 %
f05	146	54.1 %
s06	105	41.9 %
f06	182	65.4 %
s07	84	53.6 %
f07	162	53.0 %
s08	72	58.3 %
f08	164	56.1 %
s09	53	47.7 %
f09	140	64.3 %
s10	**67**	**70.1 %**

Advanced Programming

	n	passed
s02	88	26.1 %
f02	249	56.2 %
s03	65	30.8 %
f03	228	59.2 %
s04	66	43.9 %
f04	177	66.1 %
s05	70	57.1 %
f05	125	56.0 %
s06	52	44.2 %
f06	147	67.3 %
s07	53	58.5 %
f07	136	59.6 %
s08	29	51.7 %
f08	147	56.5 %
s09	22	50.0 %
f09	121	60.3 %
s10	**44**	**86.4 %**

Let us first analyze data from Introduction to programming. The long-term average (excluding spring 2010) for passed students in fall semesters is 58.5 % and in spring semesters 43.7 %. One of the reasons for the higher dropout rate in spring courses might be the student population. In spring terms most of the participants are minoring in computer science, and quite likely have weaker backgrounds for programming. As can be seen, the percentage of passed students in spring 2010 was higher than it has previously been, 70.1 % of the students starting the course passing it, the second highest pass-rate being 65.4 %. Extreme Apprenticeship seemed to bring clear benefits, especially in comparison to normal spring term results.

The trend in the Advanced programming course is similar: the average passing percentage in fall terms is 60.1 % and in spring 45.3 %, both being marginally higher than the acceptance percentages for the introductory course. This is most likely due to the fact that most of the students that fail the Introductory course do not take part in Advanced programming. The acceptance percentage in spring 2010 was 86.4 %, an all-time high in the department with a clear margin. The most natural explanation for the remarkably high passing rate is that the programming routine built during normal course implementations has been quite fragile for an average or below average student. In the Extreme Apprenticeship-based course those students who survived from the initial shock of Introduction to Programming seem to have been getting better and better all the time. With a strong routine built during the introductory course the challenging new concepts encountered in the advanced course have been rather easy to master.

6. CONCLUSIONS

The Extreme Apprenticeship presented in this paper provides a good structure for teaching skills that require building routine and learning best practices from the masters. Emphasizing scaffolding in combination with the set of values and practices yields very promising results as seen in the initial implementations with 67 and 44 students, the most important result being the significant decrease in dropout rates.

We believe that the Extreme Apprenticeship method's idea of taking continuous feedback and scaffolding to an extreme level provides enough support to also help some of the inefficient novices, who usually drop programming courses, to learn programming.

The role of relevant exercises for making learning by doing a reality is a key factor in this approach. The amount of work that a student puts into exercises can have a negative impact on motivation if the exercises do not support his learning process in a meaningful way.

The majority of the anonymous student feedback indicated that learning by doing was considered motivating and rewarding. A quote from an anonymous feedback summarizes the positive outcome of this approach: *"The best thing on the course was the amount of exercises and exercise groups and the availability of teachers. It was very rewarding to be on a course where you could understand the course content by simply working diligently. Making mistakes also helped to learn things."*

The outcome of our initial experiment was so encouraging that the same approach is currently being applied to the fall semester course with almost 200 participants.

7. ACKNOWLEDGMENTS

We thank The Head of Studies, PhD Jaakko "Gandhi" Kurhila for his support and inspiration.

8. REFERENCES

[1] O. Astrachan and D. Reed. AAA and CS 1: the applied apprenticeship approach to CS 1. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical*

symposium on Computer science education, pages 1–5. ACM, 1995.

[2] A. Bandura. *Social foundations of though and action: a social cognitive theory.* Prentice-Hall, 1986.

[3] K. Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004.

[5] S. Bergin and R. Reilly. The influence of motivation and comfort-level on learning to program. In *Sroceedings of the 17th Workshop on Psychology of Programming, PPIG'05,*, 2005.

[6] T. R. Black. Helping novice programming students succeed. *J. Comput. Small Coll.*, 22(2):109–114, 2006.

[7] R. E. Bruhn and P. J. Burton. An approach to teaching java using computers. *SIGCSE Bull.*, 35(4):94–99, 2003.

[8] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 111–122. ACM, 2007.

[9] A. Collins, J. Brown, and S. Newman. Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In *Knowing, Learning and Instruction: Essays in honor of Robert Glaser.* Hillside, 1989.

[10] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: making thinking visible. *American Educator*, 6:38–46, 1991.

[11] S. Grissom and M. J. Van Gorp. A practical approach to integrating active and collaborative learning into the introductory computer science curriculum. In *Proceedings of the seventh annual consortium on Computing in small colleges midwestern conference*, pages 95–100, USA, 2000. Consortium for Computing Sciences in Colleges.

[12] M. Hassinen and H. Mäyrä. Learning programming by programming: a case study. In *Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, pages 117–119. ACM, 2006.

[13] C. Heines. http://katas.softwarecraftsmanship.org/.

[14] K. Huffman and M. Vernoy. *Psychology in Action.* Wiley, 2003.

[15] T. Jenkins. The motivation of students of programming. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 53–56. ACM, 2001.

[16] M. Kölling and D. J. Barnes. Enhancing apprentice-based learning of java. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 286–290. ACM, 2004.

[17] M. R. Lepper. Motivational considerations in the study of instruction. *Cognition and Instruction*, 5(4):289–309, 1988.

[18] L. Lumsden. *Motivation, Cultivating a Love of Learning.* ERIC Clearinghouse on Educational Management, University of Oregon, 1999.

[19] R. Martin. Review of the Pete McBreen's book Software Craftmanship, http://www.mcbreen.ab.ca/SoftwareCraftsmanship/.

[20] R. E. C. Paul A. Kirschner, John Sweller. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, problem-based, experiental, and inquiry-based teaching. *Educational Psychologist*, 41(2):75–86, 2006.

[21] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223. ACM, 2007.

[22] D. Perkins, C. Hancock, R. Hobbins, F. Marsin, and R.Simmons. Conditions of learning in novice programmers. In *Studying the novice programmer*, pages 261–279. Lawrence Erlbaum, 1989.

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

[24] H. Roumani. Design guidelines for the lab component of objects-first cs1. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 222–226. ACM, 2002.

[25] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

[26] D. Stipek. *Motivation to Learn: From theory to practice.* Prentice Hall, 1988.

[27] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes.* Harvard University Press, Cambridge, MA, 1978.

[28] K. J. Whittington. Infusing active learning into introductory programming courses. *J. Comput. Small Coll.*, 19(5):249–259, 2004.

[29] S. Wiedenbeck, D. LaBelle, and V. Kain. Factors affecting course outcomes in introductory programming. In *Workshop on Psychology of Programming, PPIG'04*, pages 97–109, 2004.

[30] B. C. Wilson and S. Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 184–188. ACM, 2001.

[31] L. Winslow. Programming psychology - a psychological overview. *SIGCSE Bulletin*, 27:17–22, 1996.

Publication I.3

I.3

Jaakko Kurhila and Arto Vihavainen

Management, Structures and Tools to Scale up Personal Advising in Large Programming Courses

In *Proceedings of the 12th Conference on Information Technology Education (SIGITE '11)*

Management, Structures and Tools to Scale up Personal Advising in Large Programming Courses

Jaakko Kurhila and Arto Vihavainen
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ kurhila, avihavai }@cs.helsinki.fi

ABSTRACT

We see programming in higher education as a craft that benefits from a direct contact, support and feedback from people who already master it. We have used a method called Extreme Apprenticeship (XA) to support our CS1 education. XA is based on a set of values that emphasize actual programming along with current best practices, coupled tightly with continuous feedback between the advisor and the student. As such, XA means one-on-one advising which requires resources. However, we have not used abundant resources even when scaling up the XA model. Our experiments show that even in relatively large courses (n = 192 and 147), intensive personal advising in CS1 does not necessarily lead to more expensive course organization, even though the number of advisor-evaluated student exercises in a course grew from 252 to 17420. A thorough comparison of learning results and organizational costs between our traditional lecture/exercise-based course model and XA-based course model is presented.

Categories and Subject Descriptors

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

General Terms

Human Factors

Keywords

course cost, resource allocation, individual education, continuous feedback, instructional design, programming education

1. INTRODUCTION

We have organized our CS1 courses (and nowadays also other programming and data structure courses) for the last three times using a method called Extreme Apprenticeship (XA). One of the key points of XA is that it emphasizes *doing* over everything else, questioning the utility of lectures, and focuses on active teacher-student collaboration. To be more specific, there are two core values that are stressed in all course activities (adapted from the original description in [16]):

- *The craft can only be mastered by actually practicing it*, as long as it is necessary. In order to be able to practise the craft, the students need to do lots of meaningful exercises. The exercises are designed to build up both *skill* and *knowledge.*
- *Bi-directional continuous feedback makes the learning process meaningful and effective.* It is vastly more efficient if a learner receives even small signals that tell her that she is progressing *and* into the correct direction. In order to give out those signals to the learner, the advisor must be aware of the successes and challenges of the learner. In other words, the advisor must be aware of the student's activities.

The results of applying XA have been impressive in the context of our university, as the drop-out rate, pass rate and grade distribution are all improving[1]. The reasons behind the success stem from the fact that without a prior experience, learning to program has been considered a hard task to master in higher education [15, 3, 14, 12]. Personal XA-based advising sees to it that every individual student practices with tens of simple exercises already during the first week of the course, enforces active participation, and seeks to disable students' ability to procrastinate until the eve of the exam. Learning achievements become visible to the student and internal motivation goes up.

It is easy to believe that organizing XA-style personal learning is straightforward when there is a handful of eager, knowledgeable students and enough competent, hardworking teachers. However, a formal educational organization has its organizational objectives that go beyond a single CS1 teacher; often it means that there are volumes of students with varying backgrounds, i.e. no prior programming experience and relatively low intrinsic motivation toward the subject matter.

This paper describes what kind of human resources and tools are needed in order to enable XA-style education in a large CS1 course. In other words, it is a description of how

[1]Data in an unpublished manuscript in preparation [10].

XA-based education can be scaled up to meet the needs of a formal educational organization that has to serve some two hundred students in CS1. This type of research has been relatively rare and is often only available in a form of policy-level meta-discussion (see e.g. [13]). The empirical evidence behind our attempt is largely based on three separate cycles of XA-based CS1 courses[2]. The first cycle of our CS1 courses (Part I and Part II) consisted of 67 and 44 students. When we scaled up the course for Fall 2010, we applied the XA model for significantly larger (192 and 147 students) CS1 courses. In Spring 2011, we applied the XA model with no lectures.

The next section describes the XA method and establishes its value by presenting the improvement in learning results. The remaining sections discuss the issues that arise when one scales up the XA method.

2. EXTREME APPRENTICESHIP

Extreme Apprenticeship (XA) builds on Cognitive Apprenticeship [7, 8], a classic model for learning. First, the student is provided with a conceptual model of the process.

Second, students are exposed to tasks (i.e. exercises) that are to be completed under material and advisor *scaffolding.* Scaffolding refers to supporting students in a way that they are not given answers, rather, just enough hints to be able to discover the answers to their questions themselves. Scaffolding works especially well if students are in the zone of proximal development [17]: not too hard, not too easy, just able to do if properly advised.

Scaffolding is faded away when the student starts to master a task.

Practical operation of XA

Many earlier applications on programming education that rely on Cognitive Apprenticeship exist [1, 2, 4, 9]. Extreme Apprenticeship differs from these by the practical issues involved:

1. start with exercises; use small incremental exercises that ensure achievable tasks; exercises need to provide clear guidelines on how to start solving the task and when a task is considered finished
2. exercises define lecture form and content; minimize lecturing and maximize number of exercises
3. advisor must be present in a same space when student is working on the exercises
4. best up-to-date programming practices are emphasized throughout the scaffolding phase
5. students are encouraged to extend their knowledge beyond the instruction provided

Practices 3–5 pose a challenge to the resource consumption and allocation when the number of participants in a course grow. Practice 3 requires added resource consumption, and Practices 4 and 5 require competence from advisors. Issues with practices 4 and 5 are out of the scope of this paper, since we did not try to purposefully improve the competence of the advisors, even though it is a benefit for the student if the advisor is competent in versatile ways.

[2]In Fall 2010, we used XA-based approach also in our "Computer as a tool" course; in Spring 2011 XA was used partly in our CS2 course (called "Data Structures") and in a new course called "Clojure Programming". The XA-related statistics concerning these courses other than CS1 are out of scope of this paper.

2.1 Extreme Apprenticeship in CS1

Course contents

Our semester-length (6+6 weeks) CS1-type introductory Java programming course consists of two separate parts: Introduction to Programming (part I) and Advanced Programming (part II). Topics covered in both the courses are typical: assignment, expressions, terminal input and output, basic control structures, classes, objects, methods, arrays and strings; advanced object-oriented features such as inheritance, interfaces and polymorphism; the most essential features of Java API, exceptions, file I/O and GUI.

In addition to the exercises, all the study material shown in the lectures is available to students on-line as a web page but written in concise XA style. The material blends both exercises and supporting material, providing students scaffolding as they proceed.

Lectures

As the principles of XA state, lectures are not a necessity in learning to program. This is also evident in our experiments. In Spring 2010, we reduced the number of lectures from the usual 5 hours per week to 2 hours for the first part of CS1, and from 4 to 2 in the second part. In Fall 2010, the responsible lecturer was willing to reduce his lecture hours from 5 to 4 per week. The second part remained the same, 4 hours a week.

It should be noted that there is no minimum in the number lectures in XA-based education. In our Spring 2011 CS1 course, there was only one 1 hour lecture in the whole course (parts I and II combined). Yet the results from the first part are even better than our previous XA-based courses (see the stats in section 2.2).

Exercises

It is expected that students in XA-based courses use most of the time they devote to the course in active solving of programming exercises – either in the computer lab or at home, if the student feels that she has less need for scaffolding. This trains the routine and gives a constant feeling of success by achieving small goals.

For each week a set of new exercises is introduced. We have had number of exercises ranging from 15 to 40. Especially at the start of the course, most of the exercises are small and relatively straightforward. Sequentially completed small exercises combine into larger, more complex programs, which are substantially more challenging than the exercises in our traditional CS1 courses. An added benefit is that combination of smaller parts show the learner how to split a big programming task into sub-tasks.

Exercise sessions with continuous feedback

All exercise sessions need a computer lab or a room with computers with suitable software to conduct the actual programming. XA-based advising is constantly available to the students present during the exercise sessions. Anyone can enter the lab without having to reserve a specific time slot.

If a student does not have specific questions during the exercise session, the advisors are actively observing that the students are working towards the right direction with good working habits, with plenty of verbal constructive feedback.

2.2 Learning results

Comparing the outcomes of the Extreme Apprenticeship -based courses to the previous course instances in terms of percentage of passed students shows clearly that the results have improved after the introduction of XA.

Results are reported separately in the tables below for CS1 part I (Introduction to programming) and Part II (Advanced programming). The XA-based implementations are highlighted in bold face. The column titled *n* denotes the number of students in each course. The pass-rates are comparable for all the course implementations as the course exams have been kept similar[3]. Significant difference between Spring and Fall semesters that re-occur every year has previously been explained by the fact that courses on Fall semesters consist mostly of CS majors, while Spring semesters consist mostly of CS minors – in XA implementations we have not noticed considerable differences between the skills of minors and majors.

CS1 part I

	n	passed
s02	92	38.0 %
f02	332	53.6 %
s03	98	39.8 %
f03	261	64.0 %
s04	84	61.9 %
f04	211	59.2 %
s05	112	46.4 %
f05	146	54.1 %
s06	105	41.9 %
f06	182	65.4 %
s07	84	53.6 %
f07	162	53.0 %
s08	72	58.3 %
f08	164	56.1 %
s09	53	47.7 %
f09	140	64.3 %
s10	**67**	**70.1 %**
f10	**192**	**71.3 %**
s11	**80**	**73.8 %**

CS1 part II

	n	passed
s02	88	26.1 %
f02	249	56.2 %
s03	65	30.8 %
f03	228	59.2 %
s04	66	43.9 %
f04	177	66.1 %
s05	70	57.1 %
f05	125	56.0 %
s06	52	44.2 %
f06	147	67.3 %
s07	53	58.5 %
f07	136	59.6 %
s08	29	51.7 %
f08	147	56.5 %
s09	22	50.0 %
f09	121	60.3 %
s10	**44**	**86.4 %**
f10	**147**	**77.6 %**
s11	**84**	**67.1 %**

Table 1: Learning results. Courses before Spring 2010 have been organized using traditional lecture format.

3. SCALING UP EXTREME APPRENTICESHIP

It is clear that both of the core values of XA (i.e., hands-on practicing and bi-directional feedback) are inherently resource-dependent. Practicing needs a space and a computer with appropriate software; continuous feedback requires advisor input for the student.

The first value proved not to be a problem when scaling up the course. Almost every student of ours has nowadays her own laptop[4]. Most of the computer labs have been dismantled and the remaining few have been heavily underused. Therefore, it was not a problem to just book the labs for the purpose. Moreover, introductory programming does not require state-of-the-art computers or expensive software. Our lab workstations are equipped with no-cost Linux together with no-cost NetBeans as the pre-installed development environment.

The second value, continuous feedback when practicing, is clearly more difficult to scale up without a direct hit to resource consumption. Moreover, resource usage for continuous feedback is amplified significantly with XA, as there are tens of exercises for every individual. To make this difference more clear, we will present actual numbers from CS1 part I courses: For Fall 2009, our traditional approach typically had 7 exercise groups (of 25 students); every week there were 6 exercises given out. In exercise sessions, one student presented her solution in front of the group and received feedback from the teaching assistant[5]. During the 6-week course, the total number of individual feedback for exercises for all the students combined is 252 (7 * 6 * 6). In the end of the XA-based course in Fall 2010, there were a total of 17420 individually evaluated exercises.

Key solution for overcoming the challenge of resource consumption is to optimize the allocation over time dynamically by using tools, structures, and even voluntary human resources. These issues are discussed in detail next.

3.1 Dynamic coordination of XA advisors and tools

When we scaled up the XA lab for Fall 2010, we designated one of the advisors as *advisor coordinator*. This was necessary in order to manage day-to-day activities of the XA lab and allocate sufficient resources to appropriate situations. The advisor coordinator had also a final say when recruiting new advisors. The coordinator was a faculty member and received no compensation for the task.

All other advisors worked under the advisor coordinator. The rest of the advisor structure emerged implicitly. All the advisors were compensated equally even though some of the advisors stepped up more than others during the courses. So-called *apprentices*, i.e. fellow students who started to grow into the role of an advisor, emerged during the courses but were not formally recognized nor financially compensated. Some of these apprentices were recruited for the next course as proper advisors, thus enabling continuous flow of advisors to be present.

In traditional courses, teaching assistants (that correspond to advisors in XA-based courses) are compensated with 39 euros/hour. The rationale behind the relatively high pay per hour is that there is a need for preparation for hosting an exercise session. In all XA courses, the advisors are compensated only 17 euros/hr, typically 2-6 hrs per week per advisor. The rationale for the relatively low pay is that the advisors cannot prepare for the XA sessions, as the advisors encounter students' programs fresh in the lab. No advisor

[3]The low acceptance rate in CS1 part II of Spring 2011 might be explained by the number of tedious experimental exercises that were created by perhaps overly eager TAs – which in turn caused a high drop-out rate. The exam acceptance rate was high: 88 %.

[4]In addition, our department has provided new CS majors a mini-laptop computer at the very beginning of their studies. Albeit not very suitable for programming, they have been used in the XA labs.

[5]In traditional formats at out University, it is also common that when a TA asks for questions, the student do not dare to voice their concerns in a group.

complained about the lower per-hour salary, and at no point of time there has been a shortage of very competent students that want to work as advisors.

Many of the advisors are in the early stages of their studies, and their teaching experiences are limited to student tutoring at most. Some were truly novice programmers as they did not have any programming experience outside the CS1 courses. The only common denominator among the advisors is the attitude: ready to work with other students, active and eager to help. Even so, learning results and student feedback has been impressive.

Some of the advisors took a strong role very early in the course but started to fade away towards the end as they felt that their expertise was not sufficient in the latter parts of the course. However, when someone started to fade, there were always advisors who started to step up. This process was ”natural” and did not need any management.

Each advisor had the possibility to choose the most preferable time slots for him or her. On-demand service was ensured using IRC[6]. On situations where there were too many students, the advisor were able to ask for extra assistance on-line. We aimed for 1/10 advisor/student ratio in the lab. The communication tool worked also as a fast way to help and share information on problems that a specific advisor himself had faced earlier – similarly the advisors were able to ask for tips on problems they could not help to solve. In addition to IRC, the advisors used text messages and mobile phones to communicate to other advisors. Dynamical resource allocation was welcomed by the advisors.

As *ad hoc* recruitment was practiced, there was no possibility for formal training. Fellow advisors informally and implicitly trained the new advisors. In addition, a ”XA lab Manifesto” was established. The manifesto was published in a wiki and updated slightly as experience of proper advising principles grew during the courses. It simply stated few guidelines as pedagogical practices. Note that the practises were worded as personal imperatives, in order to make them more personal:

- You will advise everyone in trouble
- You will not give out solutions but guide the student as much as needed in order to nudge the student to find the solutions herself.
- Advisors do constant round-robin in the lab. Observe and comment on students' progress even if no-one asks anything.
- You will pay attention to the code style: students will learn to program according to Clean Code principles.
- Correct solutions is not enough. You need to push the style towards more understandable and maintainable code.
- Even if there is a slow moment in the lab, you as an advisor cannot sit still minding your own business!

The key issue to keep the lab records correct on a day-to-day basis was an addendum to the XA lab Manifesto, called ”Bookkeeping 101”. As every advisor understood that XA-based courses can potentially be overly expensive, we communicated clearly the no-waste approach to education and resource usage:

- Prior to course formally agreed lab-hours should be marked down
- If you are alerted to the lab when there is a need for extra advisor, lab-hours are marked down.
- If your lab-time ends but there is less than 10 students remaining and you will stay, lab-hours are marked down. *Note:* Only one advisor will mark down her hours.
- If your lab-time ends but there is less than 5 students remaining, *no advisor* will mark down her hours except in special cases.
- If you are advising in the lab for fun, e.g. when not needed or outside our normal hours, you will not mark down your hours.
- Mark your hours by the end of the day.

For bookkeeping of student exercises and allocation of advisors, we utilized online spreadsheets in Google Docs with our own macros, which allowed us to keep track of the money spent so far and the demand for advisors during specific times.

In exercise sessions, students had their completed exercises marked down to a check-list, allowing them to see the check-list filled with their completed exercises. We feel that the list played an important role in feedback; every check was an achievement. Check-lists were also updated to the course web-page at the end of every day, allowing students to see the progress of other students as well. In a way, this additional feedback for the students was nearly cost-free, as the records were kept in any case.

4. RESULTS WHEN SCALING UP XA

Key numbers about the scalability are composed into the tables 2 and 3; *participants* is the number of active students in the course; *eval. exercises* corresponds to the number of exercises that the advisors evaluated from the students; *advisors* is the number of advisors in the advisor pool in that course. The roles and thus their individual working hours varied significantly, from just few hours to 41 hours; *advisor hours* is the total hours the advisors used in total for the course; *total advisor cost* is total cost for advisor salaries during the course; *cost for lectures* is the cost for lectures[7]; *n/a* means that in the initial XA courses advisor hours were not separately tracked. It should also be noted that the advisor coordinator was taking part in the XA labs as tenured faculty developing education, so his salary is not included in XA courses, even though he was actively conducting XA-style advising in the labs.

Fall 2009 courses are traditional and thus based on lectures and exercises; all the other three course instances are full XA-based courses.

We can see from Table 2 that there is a difference in advisor cost (called TA in traditional Fall 2009 course). However, it is more than compensated by the number of exercises (17420) and contact hours (310) the advisors did for the students.

[6]In practise, any instant messaging tool, e.g. Facebook Chat, Google Talk, Skype or MSN Messenger, can be used – our advisors used mainly classic IRC as they were already using it.

[7]Lectures in CS1 have been given by tenured teachers, therefore, the cost is hypothetical but based on our actual salary table for the external teachers. In reality, the price tag could be higher as tenured lecturers often do not handle other tasks during courses.

term	f09	s10	f10	s11
participants	140	67	192	80
eval. exercises	**252**	6409	**17420**	10648
advisors	5	3	13	11
advisor hours	72	n/a	310	223
total advisor cost	2808	n/a	**5270**	3791
cost for lectures	**3861**	1544	3088	**129**

Table 2: CS1 part I: Introduction to Programming.

term	f09	s10	f10	s11
participants	121	44	147	84
eval. exercises	**252**	5056	**7349**	9961
advisors	4	7	13	11
advisor hours	72	n/a	271	166
total advisor cost	2808	n/a	**4607**	2822
cost for lectures	3861	1544	3088	0

Table 3: CS1 part II: Advanced Programming.

In addition, Spring 2011 course shows that there is a possibility to save on lectures; in CS1 part I, lectures were cut to only 1 hr (cost of 129 compared to traditional course cost 3088), and in part II, there were zero lectures. Even with minimal lecturing, the course results (Table 1) are higher than ever, even among constantly high-achieving XA-based courses. As stated before and in the principles of XA, lectures are not a necessity.

Because of the increased number of students in Fall 2010, we pre-booked 20 lab hours every week in Fall 2010. In Spring 2010, every week had 8 hours of pre-booked lab time available for the students. In our cost structure, additional lab hours did not pose an additional burden to the budget, since lab reservations can be made free-of-charge. For us, hosting a course entirely in a computer lab would be the cheapest option, as lecture halls and other seminar rooms are rented with high hourly rates. However, money saved by room allocation is not considered in the statistics above, as the rent allocation scheme is a speciality for our university.

Involuntary charity work?

Since the advisors knew the limit for resource usage and could follow the time and money spent into advising, it is possible to think that some of them started to mark down fewer-than-real hours they spent in the XA lab. We can examine the progress of weekly time spent in XA-labs for the advisors (Table 4). It is clear from the figures in Table 4 that this was not the case: the total hours by the advisors correspond to the overall course timeline. Therefore, there is no sign that the advisors felt pressure to do "charity" work, i.e. work without the pay.

	Week					
Term	1	2	3	4	5	6
f10 - CS1 part I	40	51	56	51	50	50
f10 - CS1 part II	31	45	53	28	47	67
s11 - CS1 part I	37	38,5	34,5	41,5	35,5	36
s11 - CS1 part II	28	31	32,5	28,5	23	23

Table 4: Weekly hours used by advisors.

Advisor feedback

It is expected that the advisors think that XA requires more work than being a traditional teaching assistant. However, the advisors compared their experiences in XA-style advising to traditional models as "much more rewarding" and "not perceived consuming since it feels so meaningful". Rapid, visible progress of the students was considered efficient use of advisor time. In fact, the advisor feedback revealed that the experience was so rewarding, that many advisors volunteered (or "chilled out") in the computer lab and advised the students just for fun.

We sent out a web-form to advisors to collect feedback on the experience on a five-point likert scale[8]. We measured the following dimensions: "rewarding for the assistant" (rewarding), "laborious for the assistant" (laborious), "instructive for the student" (instructive) and "timewise efficient" (efficient). In addition to the previous four dimensions, we presented a meta-question "has improved my own knowledge" (improving). We present answers from only the advisors ($n = 9$) that have been assisting in both traditional exercise sessions and XA sessions. The statistics for the five dimensions are shown in table 5.

Question	Traditional	XA
rewarding	3.22	4.44
laborious	2.66	3.11
instructive	2.88	4.55
efficient	2.44	4.66
improving	3.77	4.44

Table 5: Feedback averages using five-point likert scale when comparing traditional exercise session format with XA exercise sessions.

Table 5 clearly displays the advantage of XA exercise sessions over traditional exercise sessions. Note that all the interviewed advisors had been working in both XA style and traditional exercise sessions.

We also gathered anonymous comments from the advisors. The easy going-feeling of XA sessions is reflected in the following advisor comment.

"XA exercise sessions work as a drop-in-model. You can just walk to the lab and start scaffolding. Exercises are small for the advisor as well, which helps guiding lots of students".

Another advisor reflected students' views in a few sentences, commented on the challenges of being constructive in large groups, and pointed out that students still tend to procrastinate during the weeks.

"Traditional exercise sessions cause far more stress for all parties. As a student one spends energy due to the anxiety of possibly having to go to the front to present your solution, and to understanding the lacks and extras in the presentations from others. As a TA you have an insane judgement- and quality control-role, that cannot be handled easily in a constructive manner for the whole group. In the XA labs it causes frustration that many students want to mark down their exercises during the last days of the week."

[8] 1: strongly disagree, 3: neither agree nor disagree, 5: strongly agree

5. CONCLUSIONS

Extreme Apprenticeship provides a solid structure to organize education that aims to build good routine in programming along with good programming habits such as principles of Clean Code [11] and integrated testing [5, 6]. A key component is that there are advisors who already master part of the craft and are willing to interact with students to help them to grow into expertise. Emphasizing scaffolding in combination with the core values and the derived practices has lead to clearly improved learning results. In fact, the results in learning and the overall feeling towards programming as a tool have helped us at the department to start to re-structure a significant part of our BSc degree courses in CS to benefit from programming. It does not mean that there is a lack of more abstract or theoretical concepts; on the contrary, we have started to see that learning the abstract can benefit from hands-on programming if the student is allowed to "code and play the abstract", not just "see and hear about the abstract". Programming is a helpful tool for most of the issues in the CS education.

It is obvious that hands-on programming practice with timely and constructively helpful feedback needs resources and flexibility in arrangements. Our experiments have shown that even if it is heavy work for all the involved parties (students, teachers and administration), it is possible to receive significant benefits without using significantly more resources. We have been able to match the resources well by adding awareness and interaction between advisors using appropriate tools along with solid processes of organization. We can sum up two principles that we applied when managing XA-based advisor structure in our context — in other words, "Extreme Management":

1. *Known and visible upper boundary for resource usage.* It is vital that every person involved in resource consumption knows the absolute limit for resource use and can view the resource consumption in (semi-)real time. This alleviates the problem that there is the last part of the course going on but all the resources are already used.

2. *Maximum flexibility in organizational structures.* When there are thousands of exercises to be checked by a dozen advisors in one course, not every detail of every aspect of the course is critical. Many of the rough edges (e.g. advisor differences) balance out during the course as there is ample interaction between the advisors and students. In practice, advisors will support each other and do not need to be under explicit supervision.

6. ACKNOWLEDGMENTS

Feedback and comments from all the advisors of all the XA-based courses have been invaluable. We want to thank all the advisors (in no particular order) for making XA possible; Aurora, Pekka, Juhana, Antti, Joel, Samuli, Niko, Matti's, Jenny, Jarkko, David, Lasse, Kimmo, Sebastian, Tuomas, Martin, Janne, Joonas, Aleksi – and Thomas!

7. REFERENCES

[1] O. Astrachan and D. Reed. AAA and CS 1: the applied apprenticeship approach to CS 1. In *SIGCSE '95: Proc. 26th SIGCSE technical symposium on Computer science education*, pages 1–5. ACM, 1995.

[2] T. R. Black. Helping novice programming students succeed. *J. Comput. Small Coll.*, 22(2):109–114, 2006.

[3] R. E. Bruhn and P. J. Burton. An approach to teaching java using computers. *SIGCSE Bull.*, 35(4):94–99, 2003.

[4] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER '07: Proc. third international workshop on Computing education research*, pages 111–122. ACM, 2007.

[5] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proc. 8th annual conference on Innovation and technology in computer science education*, ITiCSE '03, pages 7–10, New York, NY, USA, 2003. ACM.

[6] H. B. Christensen. *Experiences with a Focus on Testing in Teaching*, pages 147–165. Springer-Verlag, Berlin, Heidelberg, 2008.

[7] A. Collins, J. Brown, and S. Newman. Cognitive apprenticeship: Teaching the craft of reading, writing and mathematics. In *Knowing, Learning and Instruction: Essays in honor of Robert Glaser.* Hillside, 1989.

[8] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: making thinking visible. *American Educator*, 6:38–46, 1991.

[9] M. Kölling and D. J. Barnes. Enhancing apprentice-based learning of java. In *SIGCSE '04: Proc. 35th SIGCSE technical symposium on Computer science education*, pages 286–290. ACM, 2004.

[10] J. Kurhila. Carry-on effect in extreme apprenticeship. In preparation.

[11] R. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2008.

[12] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ITiCSE-WGR '07: Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223. ACM, 2007.

[13] L. B. Resnick and M. Williams Hall. Learning organization for sustainable education reform. *J. American Academy of Arts and Sciences*, 127(4):89–118, 1998.

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

[15] H. Roumani. Design guidelines for the lab component of objects-first cs1. In *SIGCSE '02: Proc. 33rd SIGCSE technical symposium on Computer science education*, pages 222–226. ACM, 2002.

[16] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *SIGCSE '11: Proc. 42nd SIGCSE technical symposium on Computer science education*, 2011.

[17] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes.* Harvard University Press, Cambridge, MA, 1978.

Publication I.4

I.4

Hansi Keijonen, Jaakko Kurhila, and Arto Vihavainen

Carry-on Effect in Extreme Apprenticeship

In *Proceedings of the 43rd Frontiers in Education Conference (FiE '13)*

Carry-on Effect in Extreme Apprenticeship

Hansi Keijonen, Jaakko Kurhila, Arto Vihavainen
Department of Computer Science
University of Helsinki, Finland
{hkeijone, kurhila, avihavai}@cs.helsinki.fi

***Abstract*—We argue that the first undergraduate courses are the most important ones on the student's path towards becoming a computer scientist. Therefore, during 2010-2012, we have exercised extensive effort in order to improve the first-semester Computer Science (CS) courses. We have been able to use a learning-by-doing approach called the Extreme Apprenticeship (XA) method accompanied by personal advising even for courses with hundreds of students. We claim that when high demands are met with sufficient support, students learn valuable programming skills that become a foundation that carries them in their further CS courses. In this paper, we analyze how the effects of a three-year effort of renovating our introductory programming courses propagate to further studies. Compared to the control cohorts of 2007-2009, we observe a carry-on-effect caused by the XA method in student success that is visible in the per-student average accumulation of credits after 7 and 13 months after the start of studies. In addition, we can see the effect propagating to mandatory subsequent courses, even without the XA method.**

I. Introduction

It is well known that learning to program is hard [1]. As computer science is typically not taught in high schools, first-year Computer Science (CS) students experience difficulties that are manifested in high failure and drop-out rates in the first programming courses (often referred to as CS1). Improving the operation of the first programming course has been a popular topic for years in the CS education community.

A challenge in examining the improvements in education is that the examination is oftentimes either too broad or too narrow. The administrative view to teaching development emphasizes the student throughput and number of degrees, cost-effectiveness [2], faculty readiness to adopt new teaching methods (see e.g. [3], [4]) but does not elicit the true effect of the improvements in the learning within the CS1 course. On the other hand, a teacher's view and reports thereof emphasize the uniqueness of the course without inspecting the effect on the subsequent courses [5], [6]. Extending the view from a single course is important as the expertise accumulates throughout the degree.

At our department, we have exercised extensive effort on improving the first-semester Computer Science courses that our university students encounter. Due to the teaching improvements and efforts, our department has been awarded various national teaching prizes during the last decade. During the last three years, we have made effort to investigate and apply a learning-by-doing approach accompanied by personal advising even for courses with hundreds of students. The application of the method to our introductory programming courses has both increased students success rates as well as actual learning. The results are significant, as the improvements have been made in a context, where the teaching has already been praised both by the students and the nation.

We purposefully and openly press our students to immerse themselves into a mode of building a strong programming routine by deliberate practise from day one of their studies. Contrary to many other approaches, we do not seek a silver bullet that would allow our students to spend fewer hours on learning. Instead, we want our students to really put the effort into purposefully guided learning-oriented activities. We claim that when high demands are met with sufficient support, students learn valuable hidden skills that become a foundation that carries them during their further studies.

Now, after three years of applying the method to our first semester, we are ready to examine the long-term effect of the *learning* in the students' first CS1 course. It is known that "teacher-induced learning has low persistence, with three-quarters or more fading out within one year" [7]. However, we have observed a rise in the skills of the students that suggests there is a carry-on effect from the first course to subsequent courses.

In this paper, we present the data that shows that the increase in programming skills in the first course enables more students to continue on a path to become a computer scientist. We observe an effect on the success of the studies 7 and 13 months after the initial programming course, as well as the courses immediately after the first programming course.

II. Background and Context

In Finland, there are no tuition fees for anyone in studies in higher education. There are more study positions in STEM (science, technology, engineering, and mathematics) subjects in higher education than there are high school students with a suitable high school course selection, and naturally, every institution wants to recruit good students. Unfortunately Computer Science is not among the most desirable study subjects, therefore, it is not very difficult to secure a study right in CS. The entrance exam for CS studies is based on logical and analytical skills and does not require programming knowledge. The studies start with no expectation of previous knowledge on the subject.

The Department of Computer Science at the University of Helsinki has been selected as a national Centre of Excellence (CoE) in higher education twice in a row, 3 years at a time. This is a remarkable achievement for the department, since in the last round of CoE, the status was awarded only to 10 units in the whole country. The CoE status was received based on ten years of well-documented department-wide teaching improvements, such as formalized study circles, detailed and explicit learning outcome rubrics for all mandatory and steadily recurring courses, and arranging the study environment according to the so-called *constructive alignment* [8]. Thus, it is safe to say that the education and teaching provided by the department has been highly valued and in a solid form already before the advent of the latest development that is in the focus of this paper.

Contrary to the common attitude, where faculty draws away from undergraduate education in order to teach graduate courses and fulfill research demands [9], we have exercised extensive effort to improve the undergraduate education starting from the very first courses. The most recent work in this area has been an improvement to both teaching arrangements and the content of our software engineering-related courses [10].

During the improvement, we have created and started applying a pedagogical method that we call the Extreme Apprenticeship (XA) method. As the first undergraduate courses are important on a student's path towards becoming an expert computer scientist, meaningful support activities must be organized for early courses. Approaching the early student population with too much distance can be detrimental to the learning community, students and teachers alike. Therefore, one important aspect of XA is to reduce the distance between the students and the teachers. In practice, this means constant emphasis on two-way feedback (interaction) between the teachers and the students.

A. The Extreme Apprenticeship method

Extreme Apprenticeship (XA) [11] is a method of organizing programming instruction in an effective and scalable manner [12]. It is not only about *learning about expertise* but *becoming an expert* in the practiced skill, e.g. programming. XA is influenced by Extreme Programming [13], where software development best practices such as code reviews are taken to the extreme, and Cognitive Apprenticeship [14], where emphasis is put on making tacit processes visible for the students via modeling, after which the students are scaffolded as they work on the task at hand themselves.

Exercises play a crucial role in XA education. Our courses are carefully structured around collaboratively produced learning objectives and assessment criteria that are visible to the students and teachers alike. Each learning objective is covered using several of exercises, that build on top of each other in a stepwise fashion. The stepwise increment is an adaptation of test-driven development [15] and the Spiral approach in education, where students deepen their knowledge on the topic step-by-step validating their work during each step.

Exercises are designed to help students start easy and deepen their knowledge in an iterative manner. An easy start provides feelings of success and helps students achieve their comfort level. Feelings of success feed the motivation that is known to be fluctuating strongly even within a course [16]. As the students work through the easier exercises, they practice skills that have been relevant in the earlier parts in the course, as well as are introduced to new topics in a gentle fashion. As students proceed within a course, the learning objectives of the exercises start overlapping each other, and more focus is put on facilitating deliberate practise [17].

As students start their work on a task, they first build a mental model of the problem at hand e.g. via well-structured exercise design, process recordings, or during lectures. Once the modeling phase has been continued to a state, where the student feels that she is confident about working on the task, she works on the task under guidance of a more experienced instructor, e.g. a teaching assistant. The teaching assistant scaffolds the student if she needs support, and even in such cases the student is only nudged towards a direction, where she can again proceed on her own.

In our context, the students are constantly helped in computer labs by course instructors, who actively engage the students that work on the exercises; students typically receive help within minutes, depending on the time of day, and the amount of students in the labs. Scaffolding provided by the instructors and learning material are designed to help the students to reach their zone of proximal development [18]. There are tens of weekly exercises, some of which provide step-by-step guidance for completing them, mimicking the solution process that a master can utilize while solving them, while other exercises are open-ended and allow students a larger degree of freedom for designing and programming a solution for them. While the students receive support and guidance in the labs, we also provide (semi-)weekly code reviews for some of the open-ended exercises.

Although there is no limit on the amount of guidance that a student can receive, or on the amount of times that the exercises can be returned, it is of utmost importance that as soon as the student does not require scaffolding and can proceed on their own, the scaffolding is faded, i.e. the support is reduced. The cycle of modeling, scaffolding, and fading takes place several times each week as each week typically contains several learning objectives and tens of exercises; the exercise sets for each week also have a strict deadline, after which they cannot be returned. Even if a solution that the student ends up with is correct, it may still require refinement. Depending on the quality of the solution, the student may be directed to further improve her work in the lab and apply practices such as clean code [19].

In practice, the most significant differences between XA and the traditional operation from the organizational perspective are: 1) there are no lectures in XA (or if there is, the lectures serve the exercises); 2) students are encouraged and expected to use as much time as needed to master the skills (thereby different students use very varying amounts of time in the XA lab during the week). Students are free to come and go as they wish to the XA labs. With careful allocation of resources, XA does not cost more than the traditional way of organizing lecture-based education [12].

So far the results in our XA-based programming education within the programming courses have been impressive. The change from traditional (lecture-based with take-home assignments) has resulted in a statistically significant change in acceptance rates of our programming courses; the average rates of our introductory programming course and advanced programming course have increased by 32% and 37% respectively (see [20] for further details). This is a noteworthy improvement as a lot of effort was already put into the improvement for the introductory programming course.

B. *Students as voluntary TAs*

In XA-based programming courses, most of the work is typically done in computer labs, where the teaching personnel scaffolds the students that work on the exercises. Since starting to apply XA, we have observed a substantial increase in students that are willing to help others in the labs, even on a voluntary basis. As a response to the increase, we have welcomed the student teaching assistants; many of them are in a very early phase of their studies, and are participating in the teaching community even as early as during their second semester [21].

The students that participate as teachers become legitimate peripheral participants [22] of the teaching community. Having young student members as part of the teaching community is beneficial for all parties, as it may increase the retention rate [23] and create a more enjoyable learning context [24]. Between fall 2010 and spring 2013, in addition to the course faculty, we have had 93 junior teaching assistants participating in making the XA experience as positive as possible to the students in the classes. Contrary to some other laudable efforts in using students as agents of educational reform [25], we aim to have some 20% of our students involved in XA labs as TAs.

III. Study Setting

The carry-on-effect of XA-based education is studied using three different measures: (1) grade distribution in the first mandatory programming course; (2) credit accumulation per average student 7 and 13 months after the start of their studies, and (3) success in the expected study path during the first semester by examining the success in two of the subsequent courses right after the first programming course. In all of these examinations, the point of introduction of XA-based education is the year 2010.

The data used in this study is extracted from the official study records of the University of Helsinki, and contains records for students that have enrolled at the university with CS as their major subject since 2007. In total, the database extract has information on 895 students. The yearly intake of students has been aimed at 130 (except 2007 when it was 150). In practice, there is year-by-year fluctuation since a part of the accepted students do not register for CS (as they probably have succeeded in landing a more preferable study place somewhere else). "Overbooking" in the student intake is typically something around 35% but fairly difficult to predict. This is the reason for normalizing the numbers year-by-year so that we can compare the relative, not absolute changes in the results.

The application of Extreme Apprenticeship method for the first courses was started in 2010. Hence, years 2010-2012 are post-XA years, and 2007-2009 are pre-XA years in the data. Pre-XA years means a more traditional way of organizing education around a fixed number of weekly lectures, exercise sessions, and study groups. The teacher responsible for the courses Introduction to Programming and Advanced Programming has been the same during 2007-2011, while a different teacher was assigned to the courses during 2012.

A. *Grade Distribution*

Even though the first programming course has been completely revamped with the advent of XA, the paper-based final exam has been deliberately kept mandatory and as closely corresponding to the exams during the pre-XA era as possible. Therefore, the grade distribution of the course Introduction to Programming is comparable on a yearly basis. It is important to note that the grade distribution at our university is completely decidable by the teacher responsible for the course. In other words, the grades do not need to be forced into a bell curve. Changes in grade distribution can therefore truthfully reflect the changes in student skills and knowledge.

We acknowledge the fact that since XA is about heavy practice, students are likely to accumulate a stronger programming routine and other desirable qualities that are not captured by the traditional final exam. In order to emphasize the thinking and not just the doing – as Allendoerfer et al. [26] aptly put it – a paper-based final exam that requires higher-order thinking skills is a valid addition to the educational arrangements, even when XA is employed.

B. *Credit Accumulation*

Possible differences in credit accumulation were analyzed by extracting the number of computer science credits that each student had gathered in 7 and 13 months since the start of their studies. As there is always a handful of students that do not start their studies during the same year they enroll, we removed the students that had not attempted to take any courses from the analysis.

As the number of students starting their studies each year differs, the credit accumulations have been normalized based on the number of active students, i.e. students, that have at least attempted a single course. After normalization, the results are directly comparable. As the students start their studies on August 1st, the 13 month accumulation for year 2012 is not available during the time of writing this article.

C. *Early Study Path Success*

In addition to the credit accumulation for each student group, we analyze if there is any difference in study path success between students. The student cohorts are built based on the year when they took their first introductory programming course, which is considered as the first step in computer science studies, as it is mandatory for every student.

We analyze two different course pairs for each student cohort: (1) Introduction to Programming and Advanced Programming (both changed to XA after 2010), and (2) Introduction to Programming and Software Modeling (the latter has not been changed to XA). The courses are organized right after the course on Introduction to Programming in the same semester.

The method employed here resembles the research conducted by Carrell and West [27]. However, as the organization of courses and allocation of teachers is not so structured at our department and administratively collected student feedback infrequent, we cannot examine the effects of XA in such a comprehensive fashion.

IV. Data and Results

In this section, we show the data and the results extracted from the study registry. First, we discuss how the grade distribution has changed within our introductory programming course. Then, we focus on the overall credit accumulation between students that have started during different years, and finally, we consider students' early study path success.

It should be noted that there have not been other significant organizational arrangements that can interfere with the results. The required study path for BSc students has been the same from 2007 to 2012. The number of teachers has remained the same, and no differences in student intake can be evoked. However, as the yearly intake in 2007 was 150 and only 130 from 2008 to 2012, we use the year 2008 as a baseline, as one could argue that the student cohort of the year 2007 was somehow inferior to the subsequent years.

Another worthy detail is that all teachers responsible for these courses are tenured teachers, not adjunct or contingent. All of the courses have had several students as paid TAs; the number and the "quality" of TAs is comparable year-by-year.

A. Grade Distribution

The grade distribution in the introductory programming courses from 2007 to 2012 is visible in Figure 1. The areas in dark color, i.e. grade 0, depict the number of students that have failed the course, while the areas in brighter color indicate students that have passed the course. In the XA-based courses, 38.5% of the students have received the highest grade available, i.e. 5, which is indicated by the brightest color. The grade 5 has been awarded to 22.6% of the students in traditional courses. During 2007-2009, 42.3% of the students failed the course Introduction to Programming on the first try, while during 2010-2012, 28.2% of the students failed the course on the first try.

It is clearly visible that the grade distribution and pass rate has been improving. However, as we do not employ tests similar to ACT/SAT, we are not able to directly compare e.g. grade inflation [28], as is possible in several other countries. However, in our context, there are no direct or hidden incentives tied to the grade distribution, and the teacher responsible for every course instance has been a tenured teacher who also teaches many of the subsequent courses to the very same students; "letting the students off easy" would be harmful for the teacher herself.

Successful start on the study path is a valuable first step, as it is a clear signal for a student that she is doing a good job and is appropriately rewarded. A successful first step can start a virtuous cycle for the student but only if the student truly has learned the required skills. Grade inflation would be counterproductive in XA-based education.

B. Credit Accumulation

Credit accumulation describes the number of credits that students have received during an observed interval. The students are grouped based on the year when they enroll at the university and start their studies. The number of credits has been aggregated from the student groups. Table I shows number of students, sum of credits after 7 months of studying

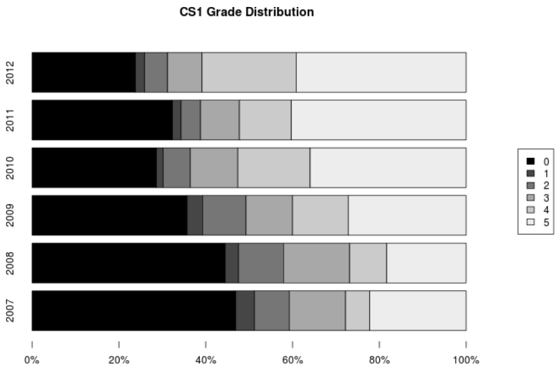


Fig. 1. Grade distribution for the course Introduction to Programming between 2007 and 2012

and credits after 13 months of studying for each student group. In addition to the sum of credits, normalized credit counts and comparison to year 2008 are also shown. The normalization is done based on the student population size, and year 2008 was chosen as a baseline as the student intake was decreased from 2007 by 20 students. Note that the data only contains students that have started their studies, i.e. at least attempted a single course.

TABLE I. Credit accumulation for student groups from different yearly intakes

Year	Students	Credits 7 (norm, scaled %)	Credits 13 (norm, scaled %)
2007	136	1681 (2237, **91.6**)	2558 (3404, **95.2**)
2008	119	1605 (2441, **100**)	2352 (3577, **100**)
2009	120	1616 (2437, **99.9**)	2686 (4051, **113.3**)
2010	136	2030 (2701, **110.7**)	3418 (4549, **127.2**)
2011	140	2287 (2957, **121.1**)	3352 (4334, **121.1**)
2012	168	3042 (3277, **134.3**)	n/a

When looking at the years 2010-2012, we observe a clear increase in the number of credits that students gain during their early studies when compared to the years 2007-2009. The higher number of freshmen in 2012 is explained by an experiment, where we utilized a massive open online course (MOOC) in programming as an entrance exam to CS studies [29]. The number of students that received a study place through "the normal way" was not influenced by the experiment.

To compare whether there is a difference between the 2007-2009 and 2010-2012 cohorts, analysis of variance (ANOVA) was conducted on the credit gains after 7 and 13 months. With three samples in both groups, there is a statistically significant difference between the groups ($p < 0.05$). In the 13 month groups, where the numbers from 2012 is missing, there is no statistically significant difference ($p = 0.062$).

This may be partially explained by the number of samples, and partially by the introduction of XA. XA was introduced during spring 2010, and some of the students that failed first programming courses during fall 2009 retook their programming courses during spring 2010. Typically, spring versions of the programming courses are populated by CS minor students, whereas fall versions are for CS major students.

In addition, we analyze the credit gains of students that attempted their studies, i.e. enrolled to at least a single class, and students that passed courses.

When analyzing the students that attempted their studies, the students in the pre-XA group gained 13.1 credits during the first 7 months (n=375, σ=11.6), while the students in the post-XA group gained 16.6 credits (n=444, σ=11.1). The two groups were also compared using an ANOVA test, which indicated that there is a statistically significant difference for the pre-XA and post-XA groups after 7 months of studies ($p < 0.01$).

When considering the credit gains after 13 months, where year 2012 has been excluded due to currently unavailable data, the pre-XA group gained 20.3 credits on average (n=375, σ=18.6), while the post-XA group gained 24.5 credits on average (n=276, σ=19.0). A statistically significant difference was observed using an ANOVA test ($p < 0.01$).

When considering students that passed courses, i.e. they have passed at least a single course, the students in the pre-XA group gained 17.0 credits during the first 7 months (n=289, σ=10.4), while the students in the post-XA group gained 19.6 credits (n=376, σ=9.3). An ANOVA test indicated that there is a statistically significant difference for the pre-XA and post-XA groups after 7 months of studies ($p < 0.01$).

When considering the credit gains after 13 months, where year 2012 has been excluded due to currently unavailable data, the pre-XA group gained 25.6 credits on average (n=297, σ=17.4), while the post-XA group gained 29.3 credits on average (n=231, σ=17.1). A statistically significant difference was observed using an ANOVA test ($p < 0.05$).

C. *Early Study Path Success*

In order to validate the effect of XA, we want to examine the mandatory first course (Introduction to Programming) and see whether the success in two mandatory subsequent courses (Advanced Programming and Software Modeling) benefits from the fact that the first course is based on XA or not. To validate the effect even further, one of the subsequent courses (Advanced Programming) is also XA-based, but the other one (Software Modeling) is not. All of these three courses are mandatory courses for every BSc student in CS, and all of the three courses are scheduled to be taken during the first semester.

In the following examination, study path success describes the student percentage that has succeeded in a specific course pair on the first attempt. The percentage for yearly course pairs is shown in Table II.

TABLE II. STUDY PATH SUCCESS WHEN MOVING FROM INTRODUCTION TO PROGRAMMING TO ADVANCED PROGRAMMING AND INTRODUCTION TO PROGRAMMING TO SOFTWARE MODELING

Year	Intr. Prg. & Adv. Prg	Intr. Prg. & SW. Modeling
2007	45.1	41.5
2008	39.2	48.8
2009	50	54.2
2010	68.5	63
2011	71.1	74.4
2012	70.3	72.2

Before XA, the year with the best success was 2009, where 50% of the students that enrolled in both Introduction to Programming and Advanced Programming passed both courses on their first attempt. A similar result is visible in the course pair Introduction to Programming and Software Modeling; 54.2% of the students that started both courses passed both on their first attempt.

The lowest scores after the introduction of XA are from the year 2010. Here 68.5% of the students passed both programming courses on their first attempt, and 63% of the students passed both Introduction to Programming and Software Modeling. However, a clear difference can be observed between the pre-XA and post-XA courses. An interesting issue is that the effect of the first course using XA appears to carry over to the subsequent course, irrespective of the use of XA in the subsequent course.

When conducting an ANOVA test for the course pair Introduction to Programming and Advanced Programming, there is a statistically significant difference ($p < 0.01$) between the pre-XA and post-XA groups. Statistically significant difference is observed also for the course pair Introduction to Programming and Software Modeling ($p < 0.05$).

V. Discussion

We have described results from a long-term study of student performance before and after applying a method called XA in our early programming courses. Our results indicate that the grade distribution, pass-rate, overall credit accumulation, and student success in staying on the desired study path have all improved after applying XA, when looking at students' performance after 7 months and 13 months of studying.

At our university, the teachers are not rewarded for performing well, nor are they punished for performing poorly. As a matter of fact, it is very difficult for a teacher to even know how they are performing, as there are no formal measures other than student grades that the teacher herself decides. The administration oversees the study progress on a larger scale but has no measures to evaluate *learning*. Many times even the formal course feedback received from the students has little impact as the teacher herself is the main actor in processing the feedback. Fortunately, in our context where teaching is valued and teachers want to be good teachers rather than bad teachers, our introductory programming courses have always received excellent feedback and been held in high esteem. Therefore, the improvements described in this paper are valuable, as they extend beyond a single course and improve an already well-functional educational arrangement.

The results and XA that are described in this paper are a part of a larger change, which was started in late 2009. Every change requires people willing to change; we have been lucky to have teachers eager to deliberately practice and hone both their skills and knowledge, and apply their knowledge fully into teaching. Communication and bi-directional feedback valued by XA requires that the teachers are on the same level as the students, as students work on their exercises: both receive feedback on what they are doing well, and what could be improved, which allows a cycle, where the courses can be aligned to match the needs of individual learners.

ACKNOWLEDGEMENTS

We gratefully acknowledge Matti Luukkainen and Arto Wikla, who have both done great work in improving the programming education at our department. We also gratefully acknowledge the grant from the Dean of Faculty of Science for this project. Part of the research is funded by the Centennial Foundation of the Finnish Technology Industries.

REFERENCES

[1] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[2] E. P. Bettinger and B. T. Long, "Does cheaper mean better? the impact of using adjunct instructors on student outcomes," *The Review of Economics and Statistics*, vol. 92, no. 3, pp. 598–613, 2010.

[3] D. L. Soldan, W. P. Osborne, and D. Gruenbacher, "Modeling the economic cost of inadequate teaching and mentoring," in *Frontiers in Education Conference (FIE), 2010 IEEE*. IEEE, 2010, pp. F3J–1.

[4] S. Fincher, B. Richards, J. Finlay, H. Sharp, and I. Falconer, "Stories of change: How educators change their practice," in *Frontiers in Education Conference (FIE), 2012*, 2012, pp. 1–6.

[5] P. Lasserre and C. Szostak, "Effects of team-based learning on a CS1 course," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 133–137.

[6] J. D. Bayliss and S. Strout, "Games as a "flavor" of CS1," in *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, ser. SIGCSE '06. ACM, 2006, pp. 500–504. [Online]. Available: http://doi.acm.org/10.1145/1121341.1121498

[7] B. A. Jacob, L. Lefgren, and D. P. Sims, "The persistence of teacher-induced learning," *Journal of Human Resources*, vol. 45, no. 4, pp. 915–943, 2010.

[8] J. Biggs, "Enhancing teaching through constructive alignment," *Higher education*, vol. 32, no. 3, pp. 347–364, 1996.

[9] D. M. Shannon, D. J. Twale, and M. S. Moore, "TA teaching effectiveness: The impact of training and teaching experience," *Journal of Higher Education*, pp. 440–466, 1998.

[10] M. Luukkainen, A. Vihavainen, and T. Vikberg, "Three years of design-based research to reform a software engineering curriculum," in *Proceedings of the 13th annual conference on Information technology education*. ACM, 2012, pp. 209–214.

[11] A. Vihavainen, M. Paksula, and M. Luukkainen, "Extreme apprenticeship method in teaching programming for beginners," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 93–98.

[12] J. Kurhila and A. Vihavainen, "Management, structures and tools to scale up personal advising in large programming courses," in *Proceedings of the 2011 conference on Information technology education*. ACM, 2011, pp. 3–8.

[13] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[14] A. Collins *et al.*, "Cognitive apprenticeship: Making things visible." *American Educator: The Professional Journal of the American Federation of Teachers*, vol. 15, no. 3, pp. 6–11, 1991.

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

[16] A. Dillon and J. Stolk, "The students are unstable! cluster analysis of motivation and early implications for educational research and practice," in *Frontiers in Education Conference (FIE), 2012*, 2012, pp. 1–6.

[17] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "The role of deliberate practice in the acquisition of expert performance." *Psychological review*, vol. 100, no. 3, p. 363, 1993.

[18] L. Vygotsky, "Mind in society," 1978.

[19] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, 2008.

[20] A. Vihavainen and M. Luukkainen, "Results from a three-year transition to the extreme apprenticeship method," *Proceedings of the 13th IEEE International Conference on Advanced Learning Technologies*, July 2013.

[21] A. Vihavainen, T. Vikberg, M. Luukkainen, and J. Kurhila, "Massive increase in eager TAs: experiences from extreme apprenticeship-based CS1," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, ser. ITiCSE '13. New York, NY, USA: ACM, 2013, pp. 123–128. [Online]. Available: http://doi.acm.org/10.1145/2462476.2462508

[22] J. Lave and E. Wenger, *Situated learning: Legitimate peripheral participation*. Cambridge university press, 1991.

[23] C. Stewart-Gardiner, "Improving the student success and retention of under achiever students in introductory computer science," *Journal of Computing Sciences in Colleges*, vol. 26, no. 6, pp. 16–22, 2011.

[24] P. E. Dickson, "Using undergraduate teaching assistants in a small college environment," in *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 2011, pp. 75–80.

[25] G. Herman, K. Trenshaw, and L.-M. Rosu, "Work in progress: Empowering teaching assistants to become agents of education reform," in *Frontiers in Education Conference (FIE), 2012*, 2012, pp. 1–2.

[26] C. Allendoerfer, M. Kim, E. Burpee, D. Wilson, and R. Bates, "Awareness of and receptiveness to active learning strategies among stem faculty," in *Frontiers in Education Conference (FIE), 2012*, 2012, pp. 1–6.

[27] S. E. Carrell and J. E. West, "Does professor quality matter? evidence from random assignment of students to professors," *Journal of Political Economy*, vol. 118, no. 3, 2010.

[28] I. Y. Johnson, "Contingent instructors and student outcomes: An artifact or a fact?" *Research in Higher Education*, vol. 52, no. 8, pp. 761–785, 2011.

[29] A. Vihavainen, M. Luukkainen, and J. Kurhila, "Multi-faceted support for MOOC in programming," in *Proceedings of the 13th annual conference on Information technology education*. ACM, 2012, pp. 171–176.

Publication II.1

II.1

Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel

Scaffolding Students' Learning Using Test My Code

In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*

Scaffolding Students' Learning using Test My Code

Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, Martin Pärtel
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ avihavai, tvikberg, mluukkai, partel }@cs.helsinki.fi

ABSTRACT

As programming is the basis of many CS courses, meaningful activities in supporting students on their journey towards being better programmers is a matter of utmost importance. Programming is not only about learning simple syntax constructs and their applications, but about honing practical problem-solving skills in meaningful contexts. In this article, we describe our current work on an automated assessment system called Test My Code (TMC), which is one of the feedback and support mechanisms that we use in our programming courses. TMC is an assessment service that (1) enables building of scaffolding into programming exercises; (2) retrieves and updates tasks into the students' programming environment as students work on them, and (3) causes no additional overhead to students' programming process. Instructors benefit from TMC as it can be used to perform code reviews, and collect and send feedback even on fully on-line courses.

Categories and Subject Descriptors

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

General Terms

Experimentation, Human Factors, Measurement

Keywords

programming, testing, automatic assessment, verification, extreme apprenticeship, situated learning

1. INTRODUCTION

We organize our programming courses using the Extreme Apprenticeship method (XA) [16]. One of the success factors in XA is that all learning-oriented activities are as "genuine" as possible, e.g. programming is done using industry strength tools while honing good programming practices. The work that students are required to do must be well defined and achievable with students' existing knowledge, especially in the early phases of a course. Early success allows students to build both self-confidence and their programming routine, which helps them to transition towards seeing more than simple syntax.

Success in the early parts of the courses relies heavily on small and relatively easy exercises, which are designed to help understand course-specific learning objectives. For example, during the first week of our CS1 course, the students practice basic input and output, and build programs that use conditionals with almost 30 different exercises [1]. As students progress in the course, the small exercises combine into bigger programs. This is to demonstrate and teach how larger programs are solved in a step-wise manner.

The students work on the exercises in computer labs where their learning is scaffolded by numerous teaching assistants (TAs) [17]. Scaffolding means providing well-timed support to the learners' learning process, so that they can achieve such learning objectives that they could not reach on their own [18]. In addition to scaffolding, the instructors assess exercises, and provide feedback on student's programming practices and program design.

Although scaffolding and bi-directional feedback is usually beneficial for both the student and the instructor, parts of scaffolding tends to be repetitive. As there are tens of students in the labs at the same time, it is hard to build a larger picture of a students' progress.

Test My Code (TMC) has been developed to reduce the amount of repetitive tasks of the TAs, i.e. exercise checking and some parts of the scaffolding, and to increase the amount of time instructors have for mentoring and supporting students. TMC also supports gathering snapshots from the students' programming processes, as well as providing long-distance support via code reviews. It can be useful for instructors and CS education researchers alike, as one can choose how it is to be utilized in a class; it can be used as an assessment service and a data gathering tool.

This paper is organized as follows. In section 2, we motivate the work on a new assessment system, after which we give a description of programming exercises in an XA context. Section 4 describes the structure of TMC, after which we describe how it can be used to scaffold students' learning. Finally, we give an overview of the performed evaluations, the impact of TMC on our courses, and lay down planned future work.

[1]See Object-Oriented Programming with Java at `http://mooc.fi`.

2. MOTIVATION FOR A NEW SYSTEM

Several of the programming courses at the University of Helsinki are organized using the Extreme Apprenticeship (XA) method, which is a modern interpretation of apprenticeship-based learning [16]. XA emphasizes students' individual effort and communication between the learner and the advisor. Core values in XA include:

- A craft can only be mastered by actually practicing it, for as long as is necessary. To be able to practice the craft, students need meaningful activities, i.e. exercises.
- Continuous feedback between the learner and the advisor. The learner needs confirmation that she is progressing *and* in a desired direction. Therefore, the advisor must be aware of the successes and challenges of the learner throughout the course.

In order for XA to properly benefit from an automatic assessment system, the system has to accommodate scaffolding as well as support the situative perspective on learning in apprenticeship-based education [3]. The perspective views the situations where knowledge is developed and later applied as highly connected, as "methods of instruction are not only instruments for acquiring skills; they also are practices in which students learn to participate" [9].

We want our students to become proficient in programming, not only to know about programming. According to the situative perspective, abstracting theory from practice does not yield transferability [5]. As a goal of our CS education is to help students on their journey towards becoming experts in their field, chosen tools and methods have to allow "participation in valued social practices" [9] of the respective professional communities. For aspiring programmers, the tools and *practices of learning* [9] in their training should be similar to those used in the software engineering industry. Industry best practices, such as code reviewing [8], has to implemented in a way that enable instructors to perform the tasks in a non-intrusive manner.

Scaffolding of students' learning has to be well-timed and not over excessive. Due to the nature of XA, accumulated knowledge of the learning of students process is continuously gathered through the bi-directional feedback of the courses. The ability to transform this knowledge iteratively into the exercises and tests of the courses has to be featured in the automated assessment system.

There exist lots of automatic systems that are designed to assess students' programs [6, 1, 11]. However, as pointed out by Ihantola et al. [11], most of them are created for a specific course or as a part of a research project, and are not made available for distribution or modification. Two of the exceptions are the Marmoset project [14], and the Web-Cat project [7], both of which are available as open source projects.

Most of the currently available assessment systems are web-based, which means that in order to solve an exercise, a student has to download a template from a web-page, solve it, and then submit it using a web-GUI. If the tests in the assessment system are built to provide feedback to the student, retrieving the feedback from a web-page causes an extra step. In addition to poorly supporting the learning of specific tools and practices, the use of a web-GUI for downloading and submitting each would cause lots of unnecessary overhead. Constantly zipping and unzipping of projects is also not something we wish to teach our students as it does not belong to the workflow of a professional software engineer.

2.1 Requirements for Test My Code

Almost all CS and IT curricula contain courses on web-development and algorithmics, both of which benefit from tools that can be utilized to support students during their learning process.

In order to create a realistic web-development environment, the students should be able to e.g. configure downloading of dependencies and deploy the same application both locally and online. Local deployment is useful e.g. as students practicing web development should also learn to debug web applications using tools such as integrated developer consoles in web browsers. In order to properly assess and scaffold good development practices, support for testing both frontend and backend functionality is of great importance.

In algorithmics courses the learning objectives are usually a mix of analysis of run-time complexity, and the creation of implementations. Many of the current assessment systems handle algorithm run-time analysis using naive run-time clocking. Although this is sufficient for most of the cases, our large-scale courses have peaks e.g. during deadlines, which can cause false negatives in the tests. Additional false negatives are caused due to the use of cloud-based virtual machines that have fluctuations in the available processing power. One approach for handling this is deterministic algorithm analysis using bytecode counting, see e.g. [12].

As an ongoing effort, Test My Code (TMC) is currently designed to

- **be as minimally intrusive as possible**; the assessment service does not introduce any additional overhead to the students' working process. TMC downloads the exercises directly to the working environment, and supports both local and server-side tests
- **support timely scaffolding within the exercises**; new goals can be made visible only after previous ones have been finished, and adding scaffolding messages to the tests is easy
- **allow awarding points for completing smaller goals**, not just for completing full exercises
- **support bidirectional feedback**; TMC supports code reviews as well as direct feedback regarding the exercises both from the students' and the instructors' perspective
- **make testing visible**; the actual testing process is made visible, which eases students into the thought of writing their own tests
- **support web-application development courses**; testing of both front-end and backend functionalities, as well as ad-hoc downloading of dependencies, is made possible using Maven[2]
- **support algorithm testing** in unstable environments

[2]`http://maven.apache.org`

From the course instructor's perspective, TMC

- **causes no additional overhead from the management perspective**; the system has a clean integration interface for reading in assessed exercises
- **allows mistakes in the exercise generation process**; if an exercise contains mistakes, updated versions can be easily published to students
- **allows honing software engineering practices for exercise developers**; generating exercises and tests does not substantially differ from work done in a normal software engineering context
- **gathers data from students' programming process** for future analysis and e.g. plagiarism detection

In addition, TMC is open source and is freely available[3].

3. TEST MY CODE

In its current state, TMC consists of several components that are organized using client-server architecture, see Figure 1. The NetBeans plugin

- retrieves and updates course exercises from an assessment server
- displays built-in scaffolding messages during the working process
- submits exercises to the assessment server
- allows giving and receiving direct feedback regarding the exercises
- gathers data from students' programming process

In addition, the plugin introduces a new menu option called TMC and three new toolbar buttons. The TMC menu contains options for changing settings (e.g. username, course, exercise directory), checking for new exercises, submitting answers, and requesting and viewing code reviews. The toolbar buttons are added for (1) running the currently modified application independently of any accidentally selected main project, (2) running local tests, and (3) submitting the solution TO the the assessment server. If the student presses the "run tests locally" button, locally available tests for the exercise are run and possible scaffolding messages are shown immediately.

The web-interface allows students to register to a course, view their statistics, and optionally submit exercises outside the IDE. Course instructors use the web interface for administrative tasks such as the creation of new courses, refreshing exercises, viewing submissions, responding to code review requests, and viewing course-related statistics.

Requests to the TMC backend are routed using one or more web servers. Each course has a git-repository that contains the course exercises. Students' submissions and information are entered into a database. Once a student submits an exercise, the exercise is verified on one of the sandbox servers that run transient user-mode Linux[4] virtual machines. Each sandbox server has an optional Maven cache for storing library dependencies for e.g. web development-related exercises.

[3]http://github.com/testmycode
[4]http://user-mode-linux.sourceforge.net/

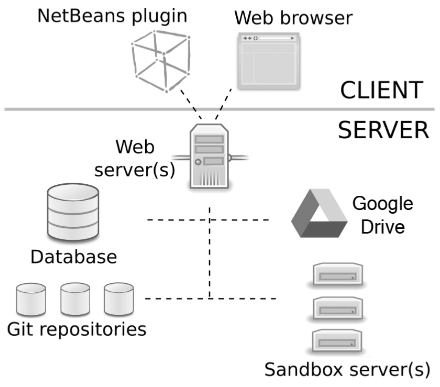


Figure 1: TMC architecture

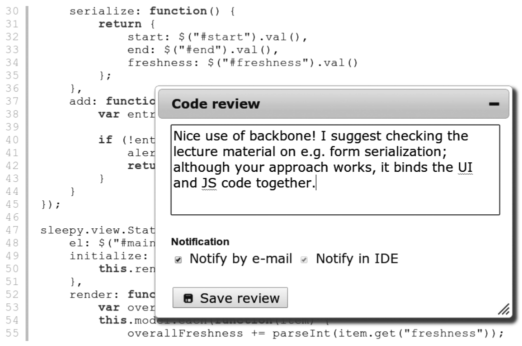
3.1 Code Reviews

Code reviews are designed to identify defects and point out improvement opportunities in the reviewed software [8]. Depending on the programming language used, some of the code reviewing can be automated. For Java typical tools are e.g. Findbugs [10] and Checkstyle, and even they fail to point out improvement opportunities in the program design and architecture. Hence, manual review is still often useful.

TMC provides code review capability in the web interface, see Figure 2. Once the students have submitted their projects, the instructors can browse the submitted code online. If students request a review for their code, the requests are visible on the TMC main page. Problems identified during a review process are communicated to the developer, who then further works on the issues.

Reviews can either be requested manually in the IDE, or the instructors can spontaneously review students' code. Once a review has been performed, the student is notified via email or directly within the IDE. As the student opens the IDE (or if the IDE is running), she sees the review comments immediately.

In our programming courses, in addition to the scaffolding in XA labs, we often perform manual code reviews at least



```
    serialize: function() {
        return {
            start: $("#start").val(),
            end: $("#end").val(),
            freshness: $("#freshness").val()
        };
    },
    add: functio
        var entr

        if (!ent
            aler
            retu
        }
    }
});

sleepy.view.Stat
    el: $("#main
    initialize:
        this.ren
    },
    render: func
        var over
        this.mode
            overallFreshness += parseInt(item.get("freshness"));
```

Code review

Nice use of backbone! I suggest checking the lecture material on e.g. form serialization; although your approach works, it binds the UI and JS code together.

Notification

☑ Notify by e-mail ☑ Notify in IDE

Figure 2: Reviewing students' code (the input window is resizeable).

for open exercises that do not impose a specific structure for the program. A single code review usually takes between 5 and 10 minutes, and each student typically receives at least one weekly review (assuming she has programmed during that week). Having manual code reviews is possible due to the way we organize our instruction [13].

3.2 Creating Exercises and Tests

Creating a new exercise and its tests starts with creating a programming project (e.g. a standard Maven project), and continues by working on the exercise using the same steps that one would while using TDD [2]. For each part of the exercise, one first creates the tests and error messages that indicate what went wrong, and afterwards add the functionality that makes the tests pass.

Once the exercise is finished, further work is needed. The finished exercise files are used as both the model solutions and the template that the students receive from TMC. The content in the version that students receive needs to be altered. Altering content is done based on comments within the project files. For example, a source file that starts with a comment `// SOLUTION FILE` is never sent to the student.

After altering the source code files is finished, the tests usually need modification. The modified version does not usually contain all the files that are referenced by the original tests. Depending on the programming language used, the tests need to be modified to tolerate conditions such as missing classes and methods. TMC provides a Java DSL that allows convenient access to student code via reflection, and provides user-friendly scaffolding messages in common error situations.

Once the tests are finished, points that the students receive for the exercise are added. Points for an exercise or a subtask are given based on annotations on test methods and classes. For example, if a test class has an annotation `@Points("007")`, the student is awarded points for the exercise "007", given that all the tests in the class pass, and no other class or test with the same annotation fails.

The exercise is published by adding it to the course git repository and refreshing the course in TMC. This causes a HTTP push event, which can be identified by the NetBeans plugin. Alternatively, the exercise (or an update to the exercise) is available the next time the student opens up the IDE.

Instructors may choose to have a separate branch for exercises that need to be tested internally before publishing. One can e.g. create a separate course for more eager students, who are willing to work as exercise testers. Once a set of exercises is tested well enough, and deemed ready for publishing, the branch is merged to the branch that is visible to the whole course. This is especially useful when performing team teaching and collaborative crafting of material.

3.3 Deterministic Profiling

Algorithm-related tests are usually assessed simply by investigating the algorithm run-time with different sized inputs. This is problematic as operating system-related tasks and e.g. JVM garbage collection tasks launch arbitrarily, causing additional load on the assessment system. This may lead to false negatives, even if the assessed algorithm is implemented perfectly. One common approach is to average the running time over a specific number of iterations. However, this simply tries to avoid the actual problem: assessing a program based on execution time.

TMC has a bytecode counting component that is inspired by ByCounter [12]. Counting bytecode instructions makes it possible to conduct repeatable benchmarks of students' algorithms. One can demand e.g. that an algorithm must run in a linear time or faster based on the size of an input.

Setting up a deterministic test needs an input for the algorithm and a method to invoke. In the following example, students need to create an algorithm for calculating the Fibonacci numbers that is linear to its input size or faster. The method must be created to a method called `fibonacci` in a class called `Fibonacci`. The method takes an integer as a parameter, and returns an Integer as output. The `IntegerImpl` is a wrapper that expects an integer as a return value from the method.

```
@Test
public void linearFibonacci() {
    IntegerImpl impl = new IntegerImpl();
    impl.setClassName("Fibonacci");
    impl.setMethodName("fibonacci");

    List<Integer> input = Arrays.asList(2, 10, 100);
    Output<Integer> output = impl.runMethod(input);
    ComplexityAnalysis.assertLinear(output);
}
```

The complexity analysis component also provides a graph view which allows easy visualization of the algorithm running times.

4. SCAFFOLDING STUDENTS WITH TMC

The driving learning method in XA is individual effort through practical work by students. This means that the exercises are the most important part of a course, and designing them is the most important task for the teacher in charge. Every instructional goal in a course is learned via working through a set of exercises designed to help building understanding on the topic at hand. Being able to solve an exercise is not enough: one must focus on both the process and quality while crafting the solution. Course instructors monitor and help students as they work on the exercises, which helps students in achieving their goals as well as provides feedback for the instructors on upcoming exercises that should be created [17].

Exercises that the students start with are usually composed of small incremental tasks, which combine into bigger programs. Incremental tasks are used to imitate a typical problem-solving process: as the students work through the tasks, they explicitly practice building software from smaller components.

The written-out thought process that was used to form the exercises constantly influence the students' programming. This provides scaffolding for learning of good programming practices, as students' work is constantly guided by the pre-performed subtask division. Exercises are intentionally written out to be as informative as possible, and often contain sample input/output descriptions or code snippets with expected outputs, which provide further support for verifying the correctness of a crafted program.

As an example of a scaffolding assignment, let us examine a sequence of exercises that demonstrates the use of methods. Just before the assignment, the course material presents the use of void methods and how a method can call another user-defined method. Before this set students solve one simple warm-up assignment involving only parameterless methods. The assignment belongs to the 2nd week of our 14-week CS1 course, and the students have practiced with loops and variables starting from week 1.

Assignment: Printing a Christmas tree

Task 1: Printing stars Modify the method `printStars` so that it prints the given amount of stars and a line break. Use the following body:

```
private static void printStars(int amount) {
   // you can print one star with System.out.print("*");
   // call it 'amount' times
}

public static void main(String[] args) {
    printStars(2);
    printStars(9);
}
```

The program should output:

```
**
*********
```

Task 2: Printing a rectangle Create a `printRectangle(int width, int height)`-method that prints a rectangle using the `printStars` method. Calling `printRectangle(17,2)` should produce the following output:

```
*****************
*****************
```

Task 3: Printing a left-aligned triangle Create a method `printLeftAlignedTriangle(int size)` that prints a triangle using the method `printStars`. Calling the new method with `3` as a parameter should produce the following output:

```
*
**
***
```

Task 4: Printing stars and whitespaces *omitted*

Task 5: Printing a right-aligned triangle *omitted*

Task 6: The tree Create a method called `xmasTree(int height)` that prints a Christmas tree using at least some of the previously defined methods. A Christmas tree consists of a triangle of given height and a stand. The stand is a single star located at the middle of the triangle bottom. The method call `xmasTree(3)`, for example, has the following output:

```
  *
 ***
*****
  *
```

While performing the steps in the above exercise, students practice creating and using methods with parameters, work constantly using a divide-and-conquer approach, and see how a simple algorithmic challenge, e.g. printing a tree, is solved.

Scaffolding within the exercises can also be used to direct the students away from bad habits such as the use of unnecessary instance variables or unclear method names. As the students work on the exercises, their workflow resembles the workflow of TDD [2] that the instructor that created the exercise previously had followed. However, in most of the cases, the tests are now pre-defined. A clear metalevel motivation for the incremental style is to guide students to follow a working process similar to that of good professional programmers: proceed in small steps and validate your code after each step [2].

The scaffolding is implemented in the tests of the exercises. The tests are written so that they help students to focus on progressing in small steps even within a single exercise: it is important to concentrate on making tests pass one by one in a meaningful order. A typical sequence might be:

1. implement class MainProgram
2. define a static method printStars(int amount)
3. ensure that the method prints the correct amount of stars
4. ensure that a newline is printed after the stars
5. define a static method printRectangle(int width, int height)
6. ensure that the method prints the correct amount of stars when called as printRectangle(1, 1)
7. ensure that printRectangle(1,1) calls printStars(1)
8. ...

4.1 Open Exercises

As any instruction should aim towards the student being able to do the problem solving themselves, it is important that scaffolding is eventually faded [4]. In our programming courses, fading is realized by using open exercises that do not enforce any specific program structure or approach.

Open assignments in early programming classes are intentionally complex enough so that programming a solution to a single file (e.g. class) causes chaotic design, but simple enough so that using an “implement a single requirement, refactor if needed” approach will eventually create a nice object design. The exercises often utilize a well-known domain (e.g. airport, airplanes or student, courses, registrations), which makes it easier to design required domain objects.

The following exercise is an example of an open exercise from week 6. In addition to the problem description, the students receive a sample input/output description, which has been left out due to space considerations.

Assignment: Bird Observations

Design and implement an observation database for a bird observer. The database contains birds, each of which have name and latin name. The database also tracks how many times a bird has been observed. The program should have a text UI and should respond to the following commands

```
add - adds a bird
observation - adds an observation
statistics - prints all birds and observations
show - prints one bird
quit - terminates the program
```

The database should store the observations into a text file for future use.

The open exercises only define how the application is supposed to work for a given user input. In early programming courses the input may be given e.g. via command line, or as system input, while in web-development courses the tests for open exercises typically monitor e.g. a given database while inputting data to input fields with specified names or ids, or require that a given REST API exists and works as desired.

5. CONCLUSIONS AND FURTHER WORK

The automated assessment system, TMC, has been successfully used in our CS1, CS2 and Web-development courses, as well as in various other courses, including MOOCs [15] in programming. We have also utilized TMC and NetBeans for younger students in a "game programming for youth" outreach course that has been created as an attempt to raise awareness towards programming. The youngest students so far have been 11, and given the challenges of learning Java at a young age, TMC itself has worked well.

As TMC can provide some of the scaffolding for the students to learn programming, it has allowed better allocation and use of resources in our courses. Instructors now spend more time on more demanding scaffolding and have time to reflect in the labs when compared to the past, where the exercises were checked manually. This has contributed to improvements in the learning results in our CS1 courses, and made the task of working as a TA a valuable learning experience [17].

The blended learning environment that TMC and XA creates resembles the genuine working environment of a software developer. Our students are immersed in this environment from day one of their studies, working with industry strength tools and pushed towards programming best practices. As TMC is not a course or topic specific tool, it can be used in a similar fashion throughout our curriculum. The following spontaneous testimonial is from a web backend development course held during fall 2012.

> TMC was great! It was great to be able to work on so many web applications. Receiving many of the configuration files with tests that said what to do aided a lot during learning. In addition, it was good that I didn't have to waste the time of TAs for showing exercises that I was confident about. TMC was very easy to install, and incredibly easy to use.

Currently TMC supports Java (and other JVM-based languages), and we are in the process of developing a more modular, language-independent, assessment backend, as well as integration to other IDEs. We are considering integrating static analysis tools such as PMD and Checkstyle with TMC, which would enable us to better address code conventions (eg. indentation, variable naming, method length and complexity) that are currently evaluated in the labs in an ad-hoc fashion. In addition, a component that flags submissions that are potentially copied from other students or model answers is under development.

We continuously wish to improve our CS education. Analyzing snapshot data from the programming process of our students, gathered by TMC, will help to improve both the exercises and the accompanied tests in our courses. This will benefit our students and teams of teachers as the learning of the students is better understood and can therefore be more effectively scaffolded.

6. ACKNOWLEDGEMENTS

We thank all the students who have worked on building TMC so far: S. Hiltunen, J. Isotalo, K. Kaltiainen, V. Knuuttila, T. Koivisto, T. Kovanen, A. Majander, P. Marjanen, J. Mynttinen, K. Nordman, M. Rannanjärvi, K. Rantanen, O. Rissanen, T. Simsiö, J. Turpeinen and K. Viiri.

7. REFERENCES

[1] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] K. Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[3] J. Brown, A. Collins, and P. Duguid. Situated cognition culture of learning. *Educational Researcher*, 18(1):32, 1989.

[4] A. Collins, J. S. Brown, and A. Holum. Cognitive apprenticeship: making thinking visible. *American Educator*, 6, 1991.

[5] A. Collins and J. G. Greeno. Situative view of learning. In V. G. Aukrust, editor, *Learning and Cognition*, pages 64–68. Elsevier Science, 2010.

[6] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Journal of Educational Resources in Compututing*, 5(3), Sept. 2005.

[7] S. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the EISTA'03*, volume 3. Citeseer, 2003.

[8] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182 –211, 1976.

[9] J. G. Greeno. Response: On claims that answer the wrong questions. *Educ. Researcher*, 26(1):5–17, 1997.

[10] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.

[11] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling.* ACM, 2010.

[12] M. Kuperberg, M. Krogmann, and R. Reussner. ByCounter: portable runtime counting of bytecode instructions and method invocations. In *Proceedings of the ETAPS'08*, 2008.

[13] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the SIGITE '11.* ACM, 2011.

[14] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with Marmoset: designing and using an advanced submission and testing system for programming courses. In *Proceedings of the ITICSE '06.* ACM, 2006.

[15] A. Vihavainen, J. Kurhila, and M. Luukkainen. Multi-faceted support for MOOC in programming. In *Proceedings of the SIGITE '12.* ACM, 2012.

[16] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the SIGCSE'11.* ACM, 2011.

[17] A. Vihavainen, T. Vikberg, M. Luukkainen, and J. Kurhila. Massive increase in eager TAs: Experiences from extreme apprenticeship-based CS1. To appear in *Proceedings of the ITiCSE'13*, July 2013.

[18] D. Wood, J. S. Bruner, and G. Ross. The role of tutoring in problem solving. *The Journal of Child Psychology and Psychiatry and Allied Disciplines*, 17(2):89–100, 1976.

Publication II.2

Jaakko Kurhila and Arto Vihavainen

A Purposeful MOOC to Alleviate Insufficient CS Education in Finnish Schools

In *Transactions on Computing Education. 15, 2, Article 10 (April 2015)*

II.2

A Purposeful MOOC to Alleviate Insufficient CS Education in Finnish Schools

JAAKKO KURHILA and ARTO VIHAVAINEN, University of Helsinki

The Finnish national school curriculum, effective from 2004, does not include any topics related to Computer Science (CS). To alleviate the problem that school students are not able to study CS-related topics, the Department of Computer Science at the University of Helsinki prepared a completely online course that is open to pupils and students in all schools in Finland. The course is a Massive Open Online Course (MOOC), as the attendance scales without an upper bound. Schools in Finland have offered the MOOC as an elective CS course for their students and granted formal school credits for completing (parts of) it. Since our MOOC is exactly the same programming course as our university-level CS1 course, we are able to use the MOOC also as a long-lasting entrance exam to the CS BSc and MSc degrees. After two spring semesters of operation, we have observed that there are school students dispersed around Finland who are ready and willing to take on a challenging programming course online, and bridging the MOOC to a full study right makes a strong incentive to keep working on the programming assignments, even without traditional teaching.

Categories and Subject Descriptors: K.3.2 [**Computer and Information Science Education**]: Computer Science Education; K.3.1 [**Computer Uses in Education**]: Distance Learning

General Terms: Design, Human Factors

Additional Key Words and Phrases: Entrance exam, high-performing students, open online course, programming education

ACM Reference Format:
Jaakko Kurhila and Arto Vihavainen. 2015. A purposeful MOOC to alleviate insufficient CS education in Finnish schools. ACM Trans. Comput. Educ. 15, 2, Article 10 (April 2015), 18 pages.
DOI: http://dx.doi.org/10.1145/2716314

1. INTRODUCTION

Ever since the world's first GSM phone call was made in Finland in 1991, Finland boasted about being a country of high technology. Because the notion of Finland as a high-tech nation was so pervasive, it was a common thought that there was no need to separately learn computing or Computer Science (CS). In the national school curriculum for basic education, effective from January 1, 2004, CS is not mentioned;

This work is in part supported by the Centennial Foundation for the Finnish Technological Industries.
(Current address) Department of Computer Science, PO Box 68, FI-00014 University of Helsinki, Finland.
Authors' addresses: J. Kurhila, University of Helsinki Open University, PO Box 53, FI-00014 University of Helsinki; email: jaakko.kurhila@helsinki.fi; A. Vihavainen, University of Helsinki, Department of Computer Science, PO Box 68, FI-00014 University of Helsinki; email: avihavai@cs.helsinki.fi. Jaakko Kurhila is currently the director of the University of Helsinki Open University. During the work described in this article, he was the head of studies at the department of computer science as well as the director of the department's center for school collaboration called Linkki, part of the Finland's Science Education Center LUMA. Arto Vihavainen is a university teacher at the department of computer science and the former coordinator for Linkki center.

DOI: http://dx.doi.org/10.1145/2716314

the only reference is to Information and Communication Technology (ICT), which was to be integrated into all existing school subjects but not as an independent subject.

A recent European Union (EU)-wide survey made it clear that ICT has not become properly integrated into school work in Finland [European Commission 2013]. When compared to other EU countries, Finnish classrooms are excellently equipped, but the facilities are heavily underused to support learning; for example, for grade 11 in schools, Finland holds the last place in the use of desktops or laptops for learning purposes. The lack of computing or CS as a separate (elective) school subject in the national basic education curriculum means that very few school students have a possibility to experience computing education. The problem persists after basic education; the most recent estimate is that only some 2% of Finnish upper secondary school students (grades 10–12) have studie programming in school [Lappi 2008].

The Finnish education system provides schools and teachers with a high degree of freedom in what elective courses to offer and how to organize teaching in classrooms. The freedom has its benefits: Schools that employ eager teachers can seek to organize computing courses and programming education. Eager teachers are still quite rare: During 3 years (2008–2010) of surveying first-semester university students starting as CS majors at our department, not a single student named an inspiring school teacher as a reason to choose CS. The same (unpublished) surveys were conducted on first-semester university students starting in mathematics. Many mathematics students did name their school teacher in mathematics as one of the key reasons for choosing to study mathematics.

Since the curriculum in Finnish schools does not include CS or related studies, and many have considered even the use of computers something that students learn on their own, even those students who are interested in computing usually do not have the study opportunities during their school years. Lack of such topics in school subjects means that many students do not know what CS is, where it can be studied, and whether studying it would suit them or not. To remedy this issue, and to bypass the troubles on a road to bring computing-related topics to the national or regional school curricula (see, e.g., Pokorny [2013] and Tucker [2010]), the Department of Computer Science at the University of Helsinki has built and offers Massive Open Online Courses (MOOCs) that can be taken by any student in any Finnish school.

Schools that choose to offer our MOOC for their students do not need to have a proficient teacher, only a teacher or a staff member (e.g., a student counsellor) who can act as a supervisor in a possible course exam. There is an additional twist in our MOOC on programming. If a student completes a required number of weekly programming tasks, she is invited to an interview. After a successful interview,she is granted a full study right (BSc and MSc) in CS at the University of Helsinki.

This arrangement has been in operation since January 2012. More than 2,000 students have participated in the MOOC, some of them also applying for the right to study. In a study conducted in 2013, students who had received the full study right via MOOC had less drop-outs when compared to students who had been selected based on a traditional entrance exam [Vihavainen et al. 2013]. The course matches exactly our introductory programming course for degree students. By offering it for school students as a flexible online course that ties directly to the CS degree, we offer students a genuine experience of what it is like to study rigorous CS. The benefit for us is that we can harvest eager students throughout the country to apply to us by promising them a secured study place while they are still at school. As an ongoing effort, we are showing the existing demand for CS education in Finnish schools and are building momentum with a goal to having enough participants for locally organized CS courses. By providing free course material, courses, and success stories, we try to stimulate national discussion on the opportunities and benefits of including CS-related topics and areas into the Finnish school curriculum.

In this article, we describe the Finnish educational system and how computing has being taught in schools over the years. At the same time, we propose a solution to the current situation, in which teachers have little opportunities to teach computing due to a lacking domain knowledge or low student enrollment. Our analysis focuses on comparing the study performance and behavior of those students who are still in school to adult students.

Throughout this text, we use the word "school" when referring to schools for basic education (grades 1–9) and general upper secondary schools (voluntary grades 10–12) in Finland. Learners in basic education are referred to as "pupils"; learners in upper secondary schools are "students." In addition, the term "MOOC" refers only to our own MOOC in programming. It is important to make this distinction because other highly publicized MOOCs employ a different mode of operation that suits a different need. Our MOOC has a distinctive pedagogical model that emphasizes actual programming, and it is formally accredited and part of the university core operation, whereas the most well known MOOCs offered mainly by various top-tier U.S. universities are not.

The term "ICT" is used to refer to the use of computers and computer applications, whereas the term "CS" means "computer science." The term "CS1" refers to a university-level introductory programming course. Whenever possible, we have cited articles that are written in English. When referring to pupils and students, we use the feminine form whenever possible.

The rest of the article is organized as follows. Section 2 describes the Finnish educational environment, Section 3 discusses MOOCs and explains the pedagogical and technical decisions behind our MOOC. Section 4 discusses results from operating the MOOC over a period of 18 months and describes how high school-aged participants in Finland perform in the course when compared to older participants. Section 5 discusses the results in the light of the limitations of the study. Finally, Section 6 concludes this article and presents further directions of development in Finland's school curriculum work in CS.

2. SCHOOLING ENVIRONMENT IN FINLAND

2.1. Educational System

There are some noteworthy issues in Finland's educational system. First, *access*: Even if a student drops out from compulsory schooling when legally allowed (after the age of 16), she is always welcome to continue her studies, even to the highest degrees available (i.e., university doctorate). In other words, even if a student chooses vocational studies as her educational path, it never blocks her possibilities to advance her education in an academic track. This is shown in Figure 1 by the horizontal arrows between vocational education and general education. Second, education is *open* for everyone. Openness means that, from elementary to university education, there are no tuition fees in Finland. Access to the highly desired study places, such as medicine, drama, and teacher education, is restricted by stringent entrance exams and multipart selection processes.

Figure 1 shows the Finnish educational system as a chart with desired yearly time frames for each educational block. In practice, students have a flexibility with time that they use frequently in upper secondary schools but especially in universities. Median graduation times are typically longer than the 3+2-year model curriculum defined by the European standard; only 49% of Finnish university students complete their BSc or MSc degree in 5.5 years [Statistics Finland 2011]. Horizontal moves between the different blocks are common in Finland because many of the starting students at the Department of Computer Science have a background in polytechnic bachelor studies (either with a finished or unfinished polytechnic degree).

The typical path to university studies is via general upper secondary education, which typically means 12 years of schooling. During those 12 years, there is only one

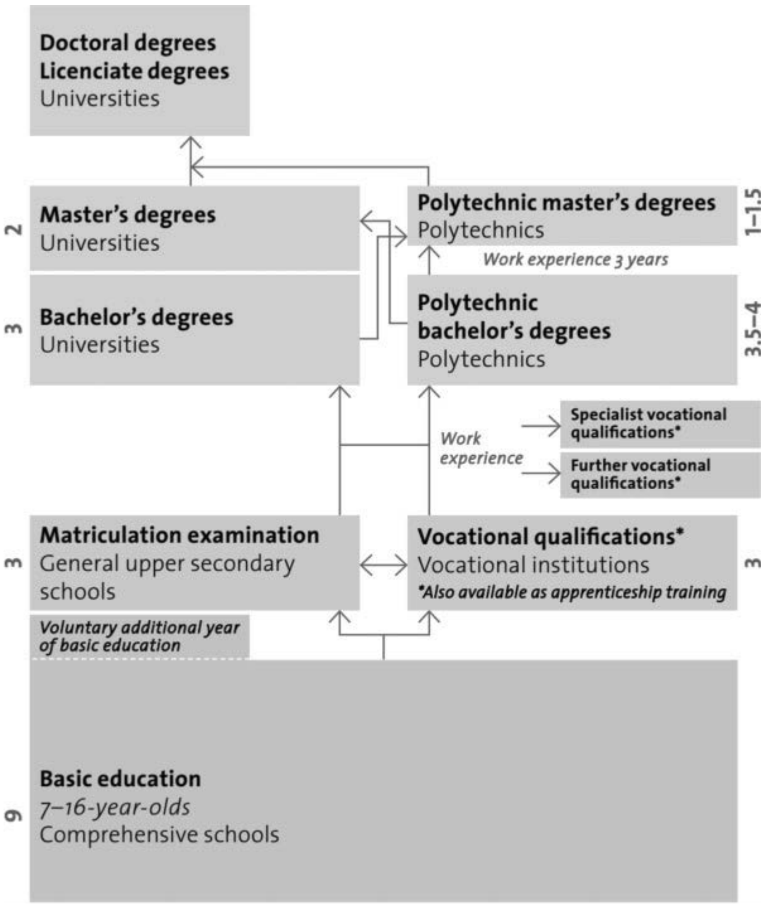

Doctoral degrees
Licenciate degrees
Universities
Master's degrees
Universities
2
Polytechnic master's degrees
Polytechnics
1–1.5
Work experience 3 years
Bachelor's degrees
Universities
3
Polytechnic bachelor's degrees
Polytechnics
3.5–4
Work experience
Specialist vocational qualifications*
Further vocational qualifications*
Matriculation examination
General upper secondary schools
3
Vocational qualifications*
Vocational institutions
*Also available as apprenticeship training
3
Voluntary additional year of basic education
Basic education
7–16-year-olds
Comprehensive schools
9

Fig. 1. Finnish educational system [Finnish National Board of Education 2011].

standardized test: the national matriculation exam after the 12th grade. Matriculation is a rigid and classic set of pen-and-paper tests that has remained relatively unchanged for decades. Students are free to choose (within clear limits) which subject exams they take from the set of offered exams and when. Even if CS-related courses are offered in a school, the matriculation exam does not contain any CS.

Since there is only one high-stakes test during the 12 years, the test has a significant impact on the activities of both students and teachers. Many students start to prepare for the matriculation exam quite early on during their 3-year upper secondary studies. Because CS is not a part of the matriculation exam, many choose not to take CS courses, even if such courses are offered by the school.

2.2. Educational Policies

The Finnish law that dictates the aims and hourly distribution of basic education states that "the aim is that the pupils learn the basics of mathematical thinking and mathematical application as well as master information and communication technologies" (1435/2001, Chapter 2, subsection 3).[1] In addition, the nationwide basic education

[1]The newest Government Decree for the aims and hourly distribution of the Finnish basic education (422/2012) was issued on June 28, 2012, and is for grades 1–6 only. It does not state anything specific

curriculum (effective from 2004 onward) states that the ability to effectively use computers and technology is a necessary skill that is to be learned at school and that the teaching should be integrated between existing subjects. However, although the tools and technology exist in classrooms, the integrative actions have failed, and the facilities remain heavily underutilized when compared to other EU countries [European Commission 2013].

2.3. Teacher Qualification

Teachers in Finland are required to have a master's degree (5 years, a total of 300 ECTS[2]). In principle, basic education teachers for grades 1–6 have a master's degree in general classroom teaching, whereas teachers for grades 7–9 and grades 10–12 have a master's degree with a major in a subject such as mathematics, biology, or English language. In addition to the major, teachers are required to complete a minor both in school pedagogy (60 ECTS including training in an actual school) and in a second school subject to be taught (60 ECTS) that can be chosen freely.

Even if computing courses are relatively rare in Finnish schools [Lappi 2008], it is still possible to study to become a teacher for computing for grades 7–9 and 10–12 in Finland. Since there is no real demand for such teachers, studying CS even as a minor is quite rare, thus creating a "chicken-and-egg" problem. Since there is no national curriculum, teacher training for CS is not well defined. And since there are not many students willing to study to become such teachers, there are not many courses in the universities that are aimed at school teachers for the purpose; most of the CS courses in a university are aimed at people who will work in industry or academia.

2.4. Sociocultural and Historical Issues of Computing in Schools

Finland has long been a country with high penetration of mobile phones, personal computers, and broadband connectivity [International Telecommunication Union 2012]. Finnish students are confident in their abilities to use technology and the internet, but, on the other hand, when compared to other EU countries, Finnish teachers show little interest in learning about information technology and the internet on their own time [European Commission 2013].

Finnish schools started to be equipped with computer labs in the 1980s. Eventually, schools were required to offer elective computing courses [Tedre and Apiola 2013], but no guidelines on the curriculum were offered by the national core curricula. In practice, the teachers in a school decided the contents for the elective courses. Courses were offered both for the upper secondary level (in grades 10–12) and at the basic education level (grades 7–9). One course is typically about 30 classroom hours in total. In the 1980s and '90s, many schools offered elementary BASIC or Pascal programming courses.

The situation has remained the same in Finnish schools to date: Some schools offer some elective courses, but no curriculum or materials are offered. In a typical school, there are no computing courses. In many schools, there is one elective course that could be labeled as "computer as a tool." Programming education has declined considerably: During the school year 1982–1983, 12% of the students participated in

about ICT, let alone CS. There is a reference to technology: "The aim of the education is a broad common knowledge, expansion and deepening of the world-view. This requires knowing the needs and emotions of humans, as well as the basics of culture, arts and literature, environment and nature, history and society, religions and worldviews, in addition to economy and technology."

[2]European Credit Transfer and Accumulation System (ECTS) is a standard for comparing the study attainment and performance of students in higher education across the EU. One academic year corresponds to 60 ECTS-credits.

programming education, whereas in 2008 only 2% of the students participated in programming education in upper secondary schools in Finland [Lappi 2008]. During the 1980s, programming was also supported by a major Finnish bank that sponsored a nationwide programming competition "Datatähti" (*Datastar*), helping the facilitation of the competition at local schools. During the 1980s and early 1990s, the Datatähti competition gathered some 4,000 participants each year. The decline was rapid following the end of the sponsoring contract: In 2010, the number of participants in the Datatähti competition was only 20 [Hyyrö et al. 2011].

It can be argued that one reason for the success of Finnish technology companies in the late '90s and early 2000 was due to the programming education in the 1980s, which allowed students to get acquainted with technology and was well-timed with the increasing number of personal computers at home. Even the currently thriving Finnish gaming industry has its roots in the late 1980s and early 1990s movement, in which students and self-taught aficionados continued to pursue program optimization and computer graphics tricks further in their free-time, contributing and competing in the so-called *demoscene*. The demoscene was particularly thriving in the Nordic countries and in Germany [Reunanen and Silvast 2009].

Currently, if computing courses are offered in schools, they are always elective, thus resembling the situation in many other countries (see, e.g., Wilson et al. [2010]). Due to a low demand, the offered courses are often cancelled prior to their start [Lappi 2008]. Even if a computing class exists and is run by the school, there is no guarantee of the educational value and content quality because the content is heavily teacher dependent; a class that is marketed as a CS class may, in the end, be about using a Word processor or Excel formulas. No guidelines are provided by government or regional officials, as there are in some other countries and regions (e.g., Hubwieser [2012]). No textbooks in the Finnish language for CS in schools are offered by common school textbook publishers. Some successful efforts for CS studies between the universities and local schools have been conducted in Finland; however, the efforts tend to be isolated and restricted in scope, such as game programming clubs or a permission for select school students to attend university classes (see, e.g., Randolph and Eronen [2007], Sutinen and Torvinen [2003], and Grandell [2005]).

3. MOOC CONTENTS AND OPERATION

MOOCs were originally defined to "integrate the connectivity of social networking, the facilitation of an acknowledged expert in a field of study, and a collection of freely accessible online resources" in a way that they may share "conventions of an ordinary course, such as a pre-defined timeline and weekly topics for consideration," but typically should carry "no fees, no prerequisites other than Internet access and interest, no predefined expectations for participation, and no formal accreditation" [McAuley et al. 2010]. MOOCs from edX, Coursera, and Udacity have attracted tens of thousands of registered participants for many different courses.

When comparing our MOOC to other contemporary programming MOOCs offered around the world, our MOOC differs in three key aspects:

(1) The course follows a pedagogical method called Extreme Apprenticeship that is particularly suitable for programming education [Vihavainen et al. 2011; Kurhila and Vihavainen 2011]. Students start programming immediately with a real-world programming environment and construct working solutions to hundreds of programming assignments during the course.
(2) The course uses stepwise assignment-driven material that is suitable for independent study. The programming environment has a purpose-built assessment solution

checker that supports the stepwise progress of the students as they work on the programming assignments [Vihavainen et al. 2013].

(3) By successfully completing the MOOC and participating in an interview, a student is granted formal credits and admitted to the university to major in CS [Vihavainen et al. 2012].

3.1. Learning Objectives and Competencies

The MOOC has explicitly stated learning objectives that are visible to all participants. The MOOC follows the structure of the CS1 at the University of Helsinki and, therefore, is split into two parts because of the quarterly term system at the University of Helsinki. The first part (5 ECTS) comprises four different main themes. The first theme is a general introduction to algorithms and control structures, the second theme is about variables and types, the third theme is about subprograms and classes, and the fourth theme is the introduction to classes, objects, and encapsulation.

In the second part (+4 ECTS), on top of the 5 ECTS, the students mainly focus on deepening their understanding of concepts in object-oriented programming, where the students learn techniques for class specification, when and why to use inheritance, and the difference between checked and unchecked exceptions and how to create and handle them. Finally, the students learn programming techniques such as splitting the application into meaningful packages and how different types of programs work: programs that ask for information, command-line interpreters, filters, and event-driven programs.

The learning objectives are presented in more detail in Appendix A.

3.2. Examination and Certification

The MOOC serves three purposes: (i) It is the compulsory programming course offered to on-campus students at the University of Helsinki, (ii) it acts as an upper secondary school course that can be freely embedded into any school curriculum, and (iii) it acts as an entrance exam. If a school chooses to offer the course to its students and wants to organize local examinations, the University of Helsinki has agreed to take care of providing and grading the exams free of charge. Schools are free to choose the number of credits they give for attending the course.

In cases (ii) and (iii), the MOOC will eventually be acknowledged as the first programming courses (9 ECTS) if the student chooses to start studying at the University of Helsinki. This means that school students can benefit twice from the MOOC because they typically get credits at school and also at the University.

3.3. Enhancing Cooperation via MOOC

There are some schools in Finland that have made agreements with local universities to allow general upper secondary school students to study CS courses even before they have gained admission to the university. Because there is no general CS curriculum in upper secondary schools, teachers have not been active in contributing to the courses within universities. Therefore, the MOOC is offered as a full CS1 course. It is beneficial for the schools: MOOC is a way to extend optional course offerings for schools becausethe complete operation is conducted outside the school.

For the University of Helsinki, the reason for making the extra effort is clear. Every university in Finland tries to attract the best students with the most suitable background. Students who are admitted via MOOC have already proved that they know what they are expected to study with us, and they have succeeded in it. This flying start is a significant bonus for us because no other university in Finland can expect any background in CS; typically, every CS1 course with newly admitted students starts with zero background expectation. In addition, the data gathered from the students as they participate in the MOOC is one of the datasets used for computing education research at the University of Helsinki.

3.4. Funding MOOC as a Long-Term Student Recruitment Vehicle

Running a MOOC on programming requires additional resources. Posting and grading exams sent to schools does not scale without limits. However, because the MOOC is an integral part of our on-campus CS1 course, the marginal cost when extending it to basic and upper secondary school students is unnoticeable for a large CS department in a country with a population of 5.5 million. Using MOOC as a recruitment vehicle is directly beneficial for our student intake process because students admitted via MOOC are less likely to retake exams and drop out from their first-year studies [Vihavainen et al. 2013].

4. RESEARCH AND RESULTS

The purpose of this research is to (i) gather background information of the MOOC students such as gender, age, and previous programming experience; (ii) measure how the participants perceive the difficulty and educational value of the MOOC; (iii) identify how the participants have proceeded in the MOOC; and (iv) study the participants' use of time and, therefore, working practices.

The MOOC in programming has been attended by 2,109 participants during the first 18 months of operation (two student intake iterations). Because the participants are not required to enter any other details than a working email address while registering, parts of this research has been conducted using a web survey. A total of 358 answers were received, which means that the numbers reported here have a 4% margin of error with a 90% confidence interval (calculated using the formula for a single proportion). In the answers, 27.6% of the respondents were 19 or younger; 1.1% of the respondents were 14 or younger. We eliminated data from participants who reportedly were 14 or younger from this study because it is unlikely that they are in an upper secondary school.

When analyzing programming behavior, the statistics have been gathered from the automated assessment and support system in our MOOC. If a student has completed a single assignment for a specific week (12 weeks total), she is considered to have participated in the course during that week. Initial participation is based on a student having completed the first assignment, creating a program that prints out *"Hello World!"*

The participants have been divided into two groups based on their age. We assume that most of the 15- to 19-year-old students are in general upper secondary schools and that most of the 20-year-old and older students are not in general upper secondary schools. This is a reasonable assumption because more than 60% of Finnish students attend general upper secondary schools, and CS studies are very rarely approached directly from vocational qualifications.

Based on these assumptions, we have formulated four research questions that examine the differences between school students and others. The research questions are presented in the following subsection headers labeled RQ.

4.1. RQ 1: Do the Groups Differ in Gender and Programming Experience?

Table I displays the participants' gender distribution. As is typical for technical study fields in Finland, the proportion of females is low in both groups. This is even more visible in the 15–19 age group, in which only 5.2% of the participants have identified themselves as female. The low percentage of female participants in the 15–19 group can be partially explained by school teachers; it is typical that a programming course is suggested to a mathematically talented male student more often than to his female counterpart [Malmivuori 2001].

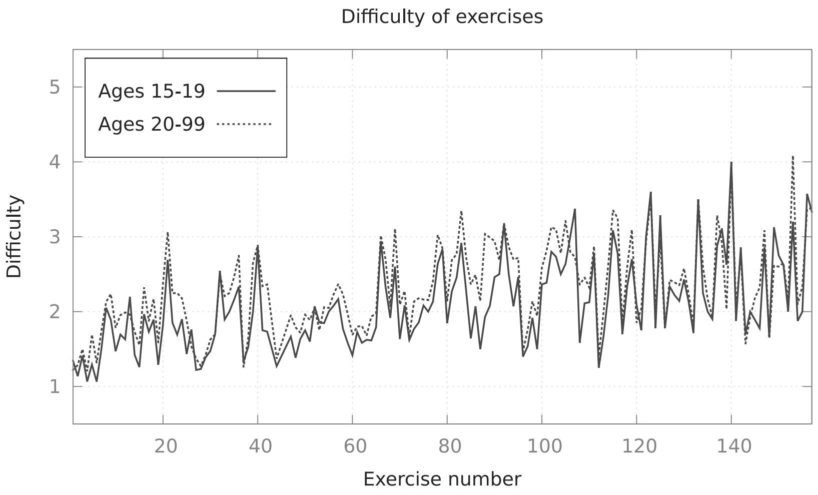


Fig. 2. Perceived difficulty of the assignments for participants grouped in two groups; participants between 15 and 19 years old, and participants 20 or older.

Table I. Gender Distribution

Group	Female	Male	Not Disclosed
15–19	5.2%	91.7%	3.1%
20–	12.8%	85.3%	1.9%

Table II. Programming Experience

Age	Programmed Previously
15–19	20.7%
20–	33.5%

Previous programming experience of participants is shown in Table II. In the survey, the participants were asked whether they had programmed before and, if yes, how many hours on their own estimate. In addition, details on which programming languages were used were asked for. Here, those participants who reported more than 10 hours of programming experience are categorized as having programmed previously. The results show that 20.7% of the under-20-year-old participants have programmed before, and, as the participants' age increases, the number of participants with existing programming experience also increases. In the group of participants 20 years and older, 33.5% have previous programming experience.

As an answer to RQ 1, we observe a clear difference in gender and programming experience between the two groups.

4.2. RQ 2: Are Difficulty and Educational Value of the Assignments Perceived Similarly by Both Groups?

Every time a participant submits a solution to a programming assignment, she has the option of answering two questions. The first question, "*How difficult was the assignment? (1: easy, 5: hard)*", measures the difficulty of the assignment on a scale from 1 to 5, and the second question, "*How much did you learn while working on the assignment? (1: nothing, 5: a lot)*" measures the educational value of the assignment. During the first 18 months, a total of 39,303 answers regarding the difficulty and educational value of the assignments have been received. The results here are averaged based on these answers, and the standard deviation has been between 0.5 and 1.2.

Figure 2 shows the difficulty of the assignments for the two groups. Although, on average, the 15–19 age group considers the assignments easier than the other group, there is no statistically significant difference ($p = 0.057$) between the groups. One

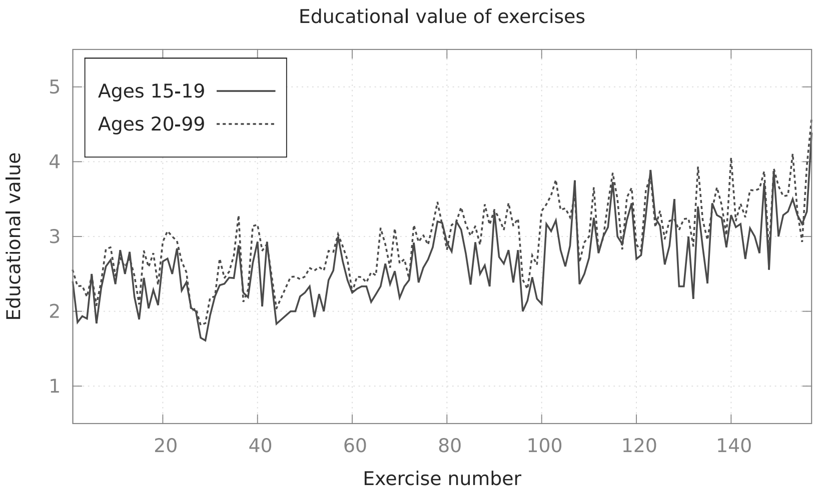


Fig. 3. Perceived educational of the assignments for participants grouped in two groups; participants between 15 and 19 years old, and participants 20 or older.

of the possible reasons is Finnish school students' high confidence with computers as tools, noted in the EU survey [European Commission 2013]. It is also possible that many of the students have been working on the assignments in a school environment with other students and possibly a teacher, hence receiving more support.

Figure 3 shows the educational value of the assignments for the two groups. The 15–19 age group considers the assignments less educational on average than the other group ($p < 0.01$). One possible explanation is the association of the educational value with learning in general; students live in an environment where they have teachers and may consider that learning is what happens when someone teaches them. On the other hand, the older participants have more experience in programming, and it can be hypothesized that they reflect their learning process on their previous learning experiences and simultaneously understand more.

When looking at Figures 2 and 3, we observe a continuous but slow ascent. The idea is that, in order to reach a goal (e.g., learn the basics of programming), it is more likely that a student is able to reach the top if she has a long but relatively easy road than if she would have to "climb steep mountains." In addition to the steady ascent, there is also a week-by-week variation. Weekly assignment sets are designed to include easy and straightforward assignments that ensure accomplishments that help the internal motivation of the student.

As an answer to RQ 2, there is no statistically significant difference in the reported difficulty between the two groups, but there is a statistically significant difference in the educational value. However, one can observe that both groups perceive the difficulty and educational value in a remarkably similar fashion.

4.3. RQ 3: Do the Groups Differ in How Far They Proceed?

Figure 4 displays the percentage of participants who have completed at least one assignment during the week, defined by the y-axis. Again, two groups are observed; those aged 15–19, and those aged 20 or over. After a week of programming, 81% of 15- to 19- year-olds are still participating, whereas for the older group the percentage is 88%. After 6 weeks, 57% of the 15- to 19-year-old students and 63% of the older participants

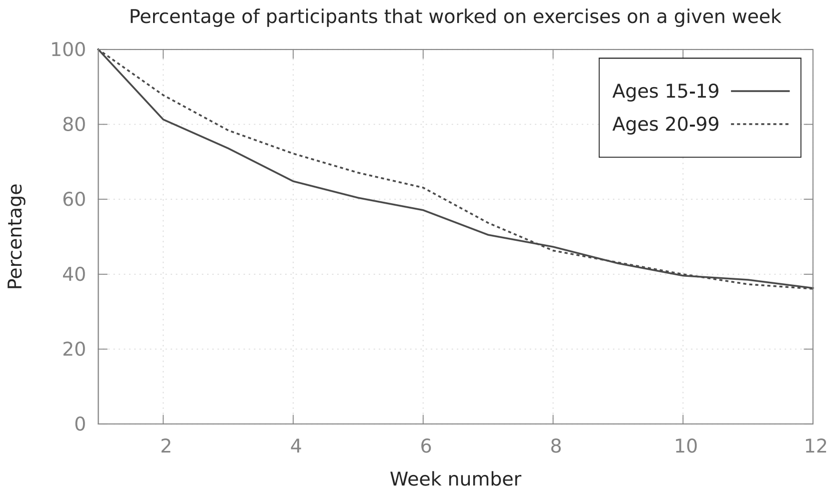


Fig. 4. The percentage of students who have worked on the assignments for a specific week grouped in two groups; participants between 15 and 19 years old, and participants 20 or older.

are still participating. During the final week of the course, 36% of the participants in both groups have solved at least one of the assignments.

One explanation for the observed early drop in the 15–19 age group is that the students may receive credits for completing parts of the course; however, because the MOOC is deliberately advertised as "free to use in schools as anyone wishes," we cannot pursue this further because the crediting schema at specific schools is not known to us. Overall, the number of participants who work on the final exercise set of the course is similar for both groups.

Overall, based on the results discussed in this subsection, we can say that the answer to RQ 3 is that there is no significant difference between the progress of the two groups in the course.

4.4. RQ 4: Does the Behavior Differ between the Two Groups?

RQ4 is set up to examine whether groups differ in their need to invest time and effort in the MOOC in order to achieve the same progress we saw in the answer for RQ3.

The working behavior of the participants is aggregated from the assignment submissions. Each submission has a timestamp, which describes the time when the assignment was submitted for automatic grading. The weekly assignment deadlines were set for 11:59 p.m. on Sundays. Figure 5 displays how the participants' programming has been split on different days of the week. For both age groups, 15–19 and 20 and over, most of the submissions are done on Sunday, which is typically a free day for all. It is also likely that the deadline has an effect on the students' work. Students tend to work more during the weekend than their counterparts and work less during the week. The age group 20 and over has a more stable working behavior (i.e., the number of daily submissions between Monday and Saturday stays in the same range), whereas the 15–19 age group works less on Tuesday and Wednesday, perhaps due to school-related schedules.

Figure 6 displays the percentage of submissions during specific hours of day during weekdays (Monday to Friday). Age group 15–19 works more during the afternoon and early evening, whereas the age group 20 and older have a more stable working schedule. Upon investigation, the observed peak between 12 and 13 for the 15–19 age group can

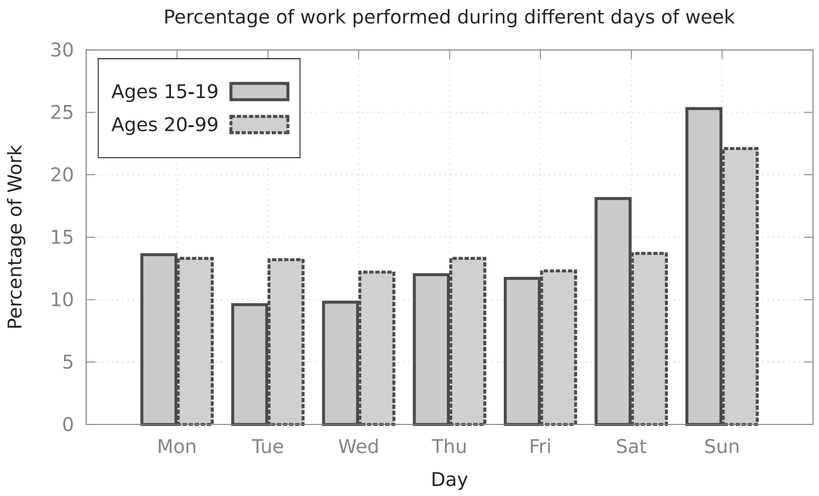


Fig. 5. Percentage of programming performed during specific days.

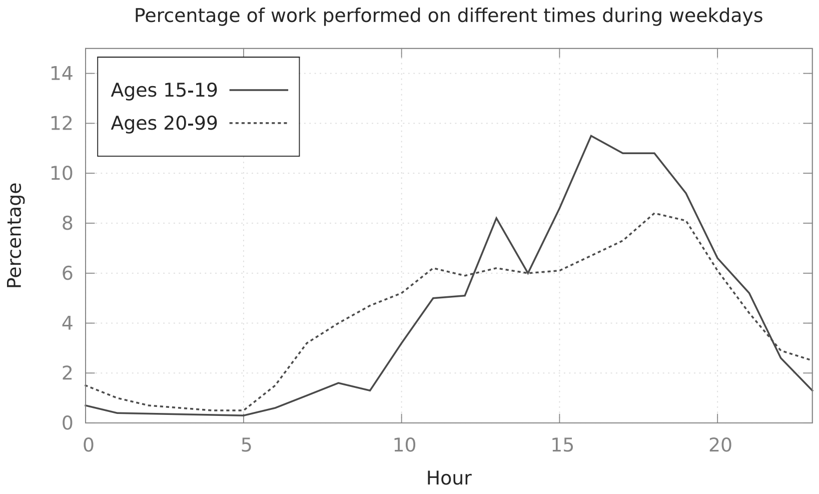


Fig. 6. Weekday hours during which the assignments have been worked on.

be explained by the lunch break at school. The more stable working schedule of the 20 and older age group can be explained by parts of them not being employed and being students at universities or polytechnics.

Figure 7 shows how the submissions are dispersed during weekends (Saturday and Sunday). Groups 15–19 and 20 and over work relatively stably during the day on weekends. No clear midnight peak that could be explained by the course deadlines can be observed.

As an answer to RQ 4, the 15–19 age group tends to work more during the weekends and less on Tuesdays and Wednesdays than the other group. The working intensity during the day resembles each other more on weekends, less on weekdays. Differences in the working habits do not lead to different level of progress (RQ3).

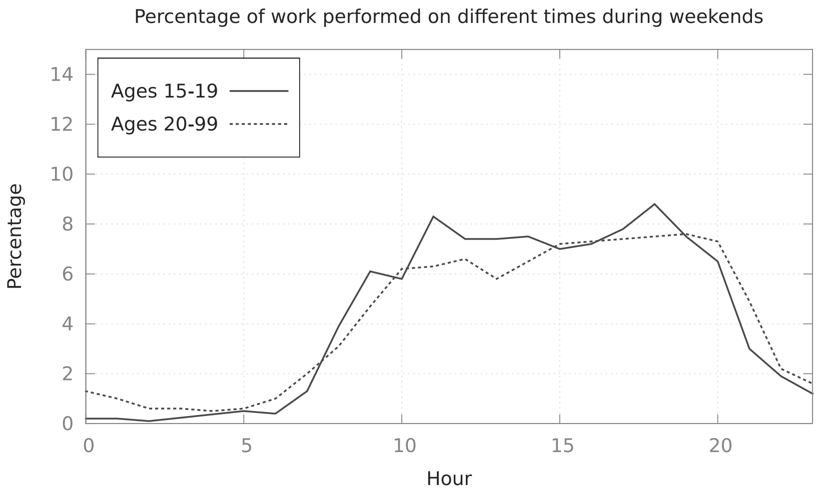


Fig. 7. Weekend hours during which the assignments have been worked on.

5. DISCUSSION: LIMITATIONS OF THE STUDY

The main limitation of the study is caused by the dataset used. We cannot deduce causal relationships within two different age groups. Study setting was designed to examine whether eager students who do not have access to school CS courses can cope with a demanding university-level programming course. The results show that age is related to some differences in working habits, previous experience, and gender balance. However, these differences do not lead to differences in the ability to produce solutions to the programming assignments.

Another limitation is that we cannot measure the effect caused by the MOOC on a national level. We have seen an increse in MOOC attendance and in student admission to CS degree studies. However, demand in the software industry, success stories of software startup businesses in mainstream media, and overall MOOC hype are likely to play a significant role in the uptake.

As seen in Table I, gender imbalance is one of the most problematic issues in our current MOOC. The intake to BSc degree in CS at our university is 18.3% female, yet in the MOOC it was as low as 5.2% among the 15–19 age group. It is obvious that unless a remedy is found, we cannot pursue strengthening the use of MOOC as a student recruitment vehicle for degree studies. On-campus degree students in CS use exactly the same MOOC as their compulsory CS1 course but do not drop out of studies any more frequently than they did before introducing the MOOC as CS1. However, our MOOC for the general population is not compulsory. Another off-putting aspect of the MOOC is that it aims to be a comprehensive and effective treatment of Java programming and makes little effort to explain *why* programming is interesting and what kind of meaningful benefits it can bring to the society. In other words, it assumes interest in programming before starting the MOOC.

We acknowledge that our MOOC is not for every school student. The course itself is a long-lasting and demanding set of tasks that requires continuous concentrated effort for completion. Therefore, it is expected that there is a strong self-select bias among the participants. It is risky to interpret the results to apply to a larger audience. For example, a version of CS0 could be a more inclusive and accessible way to introduce school students to computing concepts. In its current form and without marketing or any other push from the university, the MOOC finds an audience in schools in Finland that fits the operation well. This type of natural fit might not be always easily found (see, e.g., Feaster et al. [2011]).

6. CONCLUSION AND FUTURE WORK

In the past, having a school provide true CS education for its students in Finland has been considered a luxury because even schools with competent teachers have simply had to cancel the course offerings due to insufficient student attendance. There are some isolated projects and attempts to include school students in univerisity-level study of CS (for a recent effort, see, e.g., Lakanen and Isomöttönen [2013]). However, nationwide efforts also are needed in Finland, as demonstrated in other countries in recent years (see, e.g., Caspersen and Nowack [2013] and Bell [2014]) and even decades (see, e.g., Armoni and Gal-Ezer [2014]).

Our approach has been able to help schools that lack the chance to provide opportunities otherwise. We acknowledge that the view on CS from the MOOC is narrow because the course is specific to programming. At the same time, programming has been the first step in CS curricula for a long time (see, e.g., Joint Task Force on Computing Curricula and Society [2013, p. 41]) and is known to be a challenging topic for many [Bennedsen and Caspersen 2007; Watson and Li 2014]. Students who are admitted through the MOOC have already passed the course.

Admittedly, the attendance is still fairly small in absolute numbers. However, the yearly graduation from Finnish generic upper secondary schools is roughly 35,000 students, so the number of participants calculated based on the number of potential students (e.g., using spoken language as the baseline) means that our MOOCs are comparable to the largest MOOCs offered in the world.

Another beneficiary of the MOOC is the school student. In Finland, universities are detached from schools. Universities have complete freedom to design their admission procedures for student intake. There is no version of the U.S.- and Canada-based Advanced Placement (AP) process in Finland. A long gap between upper secondary school matriculation and successful university entrance exam is largely thought to be one of the main reasons why Finnish students are 2 years older when starting their university studies, when compared to other EU or Organization for Economic Cooperation and Development (OECD) countries [Orr et al. 2011]. The MOOC can bridge the gap from school to university studies, thus resembling AP CS [Arpaci-Dusseau et al. 2013]. Unfortunately, our MOOC is currently limited to only one university.

When comparing the upper secondary school participants in the MOOC to older participants, no significant differences exist between the populations. This means that upper secondary school students who choose to attend the course are, on average, on the same level as the other course participants. Being able to formally credit the students' progress and help the local schools with setting expectations and verifying the work plays a large part in the success of the operation. The media attention that MOOCs have gathered as a phenomenon during the past 2 years has opened up the eyes of teachers who previously would not have considered officially acknowledging students' work in an online course organized outside the school.

In Spring 2014, the Ministry of Education indicated that it has plans for introducing programming into Finnish elementary schools starting in August 2016. At the moment, the plan is to include some programming in the mathematics curriculum, but computing will not be a separate school subject in basic or upper secondary level. Schools can continue to offer elective courses.

Mathematics teachers in Finland face a problematic situation because many of them have no education to organize or facilitate school classes around programming tasks. Although the MOOC we offer is not likely to be a sufficient solution, the dislocality of the Finnish educational system and the sudden need of computing-related teacher training creates a number of openings for its use. In any case, flexible and scalable teacher training, as well as up-to-date teaching materials, are helpful in supporting teachers to meet classroom challenges.

The MOOC is the main part of our departments' outreach policy, and the operation can continue even if the number of participants increases radically. Currently, the additional resources needed are non-noticeable due to the use of cloud server technology in the assessment process. Because our MOOC is organized in Finnish, the number of participants cannot grow beyond the boundary of potentially allocable resources for a large university CS department.

APPENDIX A. LEARNING OBJECTIVES OF THE MOOC

The first part of the MOOC (5 ECTS) comprises four different main themes. The first theme is general introductions to algorithms and control structures. The learning objectives are explicitly stated, made visible to all participants, and are as follows.

—formulates simple algorithms
—explains the concept "algorithm state"
—understands how logical expressions are statements on an algorithm's state
—knows how to use basic control structures
—understands the concept of a program that asks for input data and writes output data, and can implement one
—knows the concept of arrays and can program sequential search, binary search, and some way to sort the elements of an array.

The second theme in the 5 ECTS part is about variables and types. A student:

—can use variables and write expressions using types int, double, boolean, and string
—knows the difference between primitive types and reference types
—knows the significance of assignment compatibility in programming
—understands the behaviour of formal parameters and local variables
—knows how to use classes as types and how to index an array.

The third theme is about subprograms and classes; the student can or knows how to:

—define and call subprograms, Java methods
—describe and use formal and actual parameters
—utilize a method to change the value of a parameter, if the parameter's type allows it
—overload methods and also knows how to program overloaded methods and constructors in practice.

The last theme in the first 5 ECTS package is the introduction to classes, objects, and encapsulation. A student can or knows how to:

—program instance variables and accessors
—use the technique for encapsulation and can apply it in programming
—use the concept of "object state"
—understand the lifespan of an object and how it differs from the lifespan of the local variables of methods
—pass objects as parameters and use reference types
—understand the significance of automatic garbage collection.

The latter part of the MOOC, +4 ECTS on top of the 5 ECTS, concentrates mainly on deepening the understanding of concepts in object-oriented programming. The first part of the latter package is about techniques for class specification. A student can or knows how to:

—use static and nonstatic variables as well as methods in programming
—use the possibilities and problems associated with scope rules: private attributes, package-level attributes, visibility to subclasses, public attributes.

The second theme is about inheritance. A student can or knows how to:

—use the relationship between superclass and subclass and how to program subclasses
—use the value of encapsulation of inherited fields
—understand that constructors are not inherited and what the consequences are, and can take this into consideration when programming
—override inherited methods and inherited fields.
—use (abstract or nonabstract) superclass- or interface type variables on reference-type values, and can thus implement generic classes and methods; understands polymorphism
—understand what kind of additions inheritance brings to the scope rules.

The third theme is about handling of exceptions. A student knows:

—different methods for handling exceptions.
—the principle of checked and unchecked exceptions, and can create a program where exceptions are handled at the Exception level.

The last theme is about programming techniques. A student knows or can:

—use primitive types and their assignment compatibility rules, as well as explicit type changes from broader to narrower primitive type.
—explain the function of a simple recursive method
—understand the principle of packages
—use source material to create programs that can read and write text files
—use a few generic collection classes, understand their concept, and can use material to program with them
—understand different principles for how programs work: programs that ask for information, command-line interpreters, filters, event-driven programs; in addition, also construct programs with the three first-mentioned methods.

ACKNOWLEDGMENTS

The authors wish to thank the Department of Computer Science for supporting the creation and maintenance of the MOOC. The rest of the Agile Education Research group has been an integral part of this effort and provided us valuable feedback for this manuscript. We wish to thank all the reviewers for their suggestions and feedback that helped to focus and improve the article significantly. We also thank the voluntary support personnel in the online community at #mooc.fi that motivates and helps the students in our MOOC. Two separate grants supporting the creation of MOOCs for upper secondary and basic education from the Centennial Foundation of the Finnish Technological Industries are gratefully acknowledged.

REFERENCES

Michal Armoni and Judith Gal-Ezer. 2014. High school computer science education paves the way for higher education: the Israeli case. *Computer Science Education* e-publication (2014), 1–22. DOI: http://dx.doi.org/10.1080/08993408.2014.936655

Andrea Arpaci-Dusseau, Owen Astrachan, Dwight Barnett, Matthew Bauer, Marilyn Carrell, Rebecca Dovi, Baker Franke, Christina Gardner, Jeff Gray, Jean Griffin, Richard Kick, Andy Kuemmel, Ralph Morelli, Deepa Muralidhar, R. Brook Osborne, and Chinma Uche. 2013. Computer science principles: Analysis of a proposed advanced placement course. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, NY, 251–256. DOI: http://dx.doi.org/10.1145/2445196.2445273

Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY. 999133.

Tim Bell. 2014. Establishing a nationwide CS curriculum in new zealand high schools. *Communications of the ACM* 57, 2 (Feb. 2014), 28–30. DOI:http://dx.doi.org/10.1145/2556937

Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *SIGCSE Bulletin* 39, 2 (June 2007), 32–36. DOI:http://dx.doi.org/10.1145/1272848.1272879

M. E. Caspersen and P. Nowack. 2013. Computational thinking and practice - a generic approach to computing in danish high schools. In *Proceedings of the 15th Australasian Computing Education Conference (ACE 2013) (CRPIT)*, Angela Carbone and Jacqueline Whalley (Eds.), Vol. 136. ACS, Adelaide, Australia, 137–143. http://crpit.com/confpapers/CRPITV136Caspersen.pdf.

European Commission. 2013. Survey of Schools: ICT in Education. Benchmarking Access, Use and Attitudes to Technology in Europe's Schools. Retrieved from http://ec.europa.eu/digital-agenda/en/news/survey-schools-ict-education.

Yvon Feaster, Luke Segars, Sally K. Wahba, and Jason O. Hallstrom. 2011. Teaching CS unplugged in the high school (with limited success). In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE'11)*. ACM, New York, NY, 248–252. DOI:http://dx.doi.org/10.1145/1999747.1999817

Finnish National Board of Education. 2011. Education in Finland. Retrieved from http://www.oph.fi/download/124278_education_in_finland.pdf.

Linda Grandell. 2005. High school students learning university level computer science on the web—a case study of the DASK-model. *JITE* 4 (2005), 207–218.

Peter Hubwieser. 2012. Computer science education in secondary schools – the introduction of a new compulsory subject. *Transactions in Computer Education* 12, 4, Article 16 (Nov. 2012), 41 pages. DOI:http://dx.doi.org/10.1145/2382564.2382568

Heikki Hyyrö, Erkki Mäkinen, Timo Poranen, and Antti Laaksonen. 2011. Koululaisten tietotekniikkakilpailut Suomessa. *Tietojenkasittelytiede* 33 (Dec. 2011), 27–42.

International Telecommunication Union. 2012. Measuring the Information Society. Retrieved from http://www.itu.int/en/ITU-D/Statistics/Documents/publications/mis2012/MIS2012_without_Annex_4.pdf.

Jaakko Kurhila and Arto Vihavainen. 2011. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 Conference on Information Technology Education (SIGITE'11)*. ACM, 3–8. DOI:http://dx.doi.org/10.1145/2047594.2047596

Antti-Jussi Lakanen and Ville Isomöttönen. 2013. High school students' perspective to university CS1. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'13)*. ACM, New York, NY, 261–266. DOI:http://dx.doi.org/10.1145/2462476.2465585

Linnea Lappi. 2008. Ohjelmoinnin opetus Suomen lukioissa (Programming Education in Schools, in Finnish). *Academy of Finland National Science Competition for Upper Secondary Schools (Viksu), 6th best work in 2008. Valkeakoski, Finland: Valkeakosken lukio* (2008).

Marja-Liisa Malmivuori. 2001. *The Dynamics of Affect, Cognition, and Social Environment in the Regulation of Personal Learning Processes: The Case of Mathematics* (PhD thesis). Retreived from http://ethesis.helsinki.fi/julkaisut/kas/kasva/vk/malmivuori/.

Alexander McAuley, Bonnie Stewart, George Siemens, and Dave Cormier. 2010. The MOOC Model for Digital Practice. Retreived from http://davecormier.com/edblog/wp-content/uploads/MOOC_Final.pdf.

Dominic Orr, Christoph Gwosc, and Nicolai Netz. 2011. *Social and Economic Conditions of Student Life in Europe. Synopsis of Indicators. Final report. Eurostudent IV 20082011.* W. Bertelsmann Verlag, Bielefeld. Retrieved from http://www.eurostudent.eu/download_files/documents/Synopsis_of_Indicators _EIII.pdf.

Kian L. Pokorny. 2013. What will they know? Standards in the high school computer science curriculum. *Journal of Computer Science Collection* 28, 5 (May 2013), 218–225. http://dl.acm.org/citation.cfm?id=2458569.2458616.

Justus Joseph Randolph and Pasi Eronen. 2007. Developing the learning door: A case study in youth participatory program planning. *Evaluation and Program Planning* 30, 1 (2007), 55–65.

Markku Reunanen and Antti Silvast. 2009. Demoscene platforms: A case study on the adoption of home computers. In *History of Nordic Computing 2*, John Impagliazzo, Timo Järvi, and Petri Paju (Eds.). IFIP Advances in Information and Communication Technology, Vol. 303. Springer, Berlin, 289–301. DOI:http://dx.doi.org/10.1007/978-3-642-03757-3_30

Statistics Finland. 2011. Official Statistics of Finland (OSF): Progress of studies. Retrieved from http://www.stat.fi/til/opku/2011/opku_2011_2013-03-20_tie_001_en.html.

Erkki Sutinen and Sirpa Torvinen. 2003. The candle scheme for creating an on-line computer science program – experiences and vision. *Informatics in Education* 2, 1 (2003), 93–102. Available in http://www.vtex.lt/informatics_in_education/htm/INFE009.htm.

Matti Tedre and Mikko Apiola. 2013. Three computing traditions in school computing education. In *Improving Computer Science Education*, Djordje Kadijevich, Charoula Angeli, and Carsten Schulte (Eds.). Routledge, 100–116.

Allen B. Tucker. 2010. K-12 computer science: Aspirations, realities, and challenges. In *Proceedings of the 4th International Conference on Informatics in Secondary Schools - Evolution and Perspectives: Teaching Fundamentals Concepts of Informatics (ISSEP'10)*. Springer-Verlag, Berlin, 22–34. DOI:http://dx.doi.org/10.1007/978-3-642-11376-5_3

Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. 2012. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th Annual Conference on Information Technology Education (SIGITE'12)*. ACM, 171–176. DOI:http://dx.doi.org/10.1145/2380552.2380603

Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. 2013. MOOC as semester-long entrance exam. In *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education (SIGITE'13)*. ACM, New York, NY, 177–182. DOI:http://dx.doi.org/10.1145/2512276.2512305

Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)*. ACM, 93–98.

Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'13)*. ACM, New York, NY, 117–122. DOI:http://dx.doi.org/10.1145/2462476.2462501

Christopher Watson and Frederick W. B. Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, NY, 39–44. DOI:http://dx.doi.org/10.1145/2591708.2591749

Cameron Wilson, Leign Ann Sudol, Chris Stephenson, and Mark Stehlik. 2010. Running on empty: The failure to teach K-12 computer science in the digital age. Association for Computing Machinery. Retrieved from http://www.acm.org/runningonempty/fullreport2.pdf.

Received April 2013; revised December 2014; accepted December 2014

Publication II.3

Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila

MOOC as Semester-long Entrance Exam

In *Proceedings of the 14th ACM SIGITE Conference on Information Technology Education (SIGITE '13)*

MOOC as Semester-long Entrance Exam

Arto Vihavainen, Matti Luukkainen, Jaakko Kurhila
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
Fi-00014 University of Helsinki
{ avihavai, mluukkai, kurhila }@cs.helsinki.fi

ABSTRACT

MOOCs (massive open online courses) became a hugely popular topic in both academic and non-academic discussions in 2012. Many of the offered MOOCs are somewhat "watered-down versions" of the actual courses given by the MOOC professors at their home universities. At the University of Helsinki, Department of Computer Science, our MOOC on introductory programming is exactly the same course as our first programming course on campus. Our MOOC uses the Extreme Apprenticeship (XA) model for programming education, thus ensuring that students are proceeding step-by-step in the desired direction. As an additional twist, we have used our MOOC as an entrance exam to studies in University of Helsinki. In this paper, we compare the student achievement after one year of studies between two cohorts: the MOOC intake (n=38) and the intake that started their studies during the fall (n=68). The results indicate that student achievement is at least as good on the MOOC intake when compared to the normal intake. An additional benefit is that the students admitted via MOOC are less likely to drop out from their studies during their first year.

Categories and Subject Descriptors

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

General Terms

Experimentation

Keywords

entrance exam, admission, first-year experience,
student achievement

1. INTRODUCTION

MOOCs or massive open online courses have been a source for an intense debate recently in academia, both in administration and among teachers (see e.g. [5]). MOOCs come in a variety of forms; however, most of the current high-profile MOOCs tend to be based on short lectures (8-12 min videos, animations and screencasts) interspersed with quizzes that are used to keep up the students' attention to the learning material[1]. A key issue in MOOCs is to facilitate and allow massive attendance.

MOOCs have been aptly described as "textbooks on steroids" [6]. In other words, the students that are successful in MOOCs tend to be autodidacts, to the extent that e.g. more than 70% of the starting MOOC students already have an undergraduate or postgraduate degree [13].

Our MOOC at the University of Helsinki Department of Computer Science differs from typical MOOCs in two key aspects [16]:

- Students start by installing a real-world programming environment and start to program immediately. All learning materials are built to support hands-on programming. The emphasis is heavily on a learning process that allows and requires the learners to produce working solutions. There are hundreds of programming assignments that the students are expected to construct during the course.
- By successfully completing the MOOC and participating in an interview, a student is granted admission to the university to major in Computer Science.

In Finland, the students choose their major before entering a university, making the decision often based on a relatively vague idea of the area and whether the studies suit the student. Using a traditional entrance exam as a way to select students provides insufficient results[2] as some of our first-year students fail to succeed in e.g. the very first required programming courses, effectively forcing them to seek another study.

The most important part of MOOC as semester-long entrance exam is the *process of learning to program*, during which the student sees if CS/IT is the desired area of study for her. Successfully completing the MOOC provides us and the student herself the evidence that shows that she has the aptitude for CS/IT.

[1]A notable exception are so-called connectivist MOOCs that rely more on facilitated discussions among networked learners [14, 7, 12].

[2]Attempts to pinpoint identifiable markers for aptitude to succeed in CS/IT studies have yielded non-conclusive results (see e.g. [10, 1, 15, 8]), making it impossible to derive a set of markers for revising the traditional entrance exam.

In practice, our MOOC is exactly the same course as the entry-level programming course in our university. This in itself acts as a validation measure to see whether a student is able to handle the first and often the most challenging courses.

In the first 18 months of operation, our MOOC has had 2109 participants, from which some 200 students have applied for a study position. The MOOC in programming was first used as an entrance exam in spring 2012. In addition to the new admission path via MOOC, the traditional admission procedure was kept intact and also offered to high-school students.

In this paper, we compare the success of students admitted via a MOOC to the traditional entrance exam-based intake. As our work on using MOOCs as an entrance exam to university studies has been active only for a short while and we are still adjusting the level that we require from the MOOC participants for them being admitted, we have deliberately chosen not to include statistical analysis to avoid drawing premature conclusions at this stage.

2. EDUCATIONAL SYSTEM IN FINLAND

Before starting undergraduate higher education in Finland, students typically have 12 years of schooling. During those 12 years, there is only one standardized test: the matriculation exam after the 12th grade. Major subject is chosen before starting the University studies.

Universities can use the results of the matriculation exam to grant study rights. However, most study disciplines in the universities use the matriculation exam results only as a small addition to university- and subject-specific entrance exams, especially in highly desirable subjects. In STEM subjects, the admission is typically more generous due to the lack of applicants. Admission can be granted based on either 1) solely the entrance exam, 2) solely the matriculation exam, or 3) a combination score from entrance exam and matriculation exam. At the Univ. Helsinki Dept of Computer Science most of the admitted students have taken the entrance exam (402 applicants in 2013) and received some extra points based on their matriculation exam. Entrance exams for CS/IT – and other subjects as well – are classic pen-and-paper tests conducted in a lecture hall under strict surveillance so that candidates are using only their brains to answer the exam questions.

As computer science (computing, or any IT-related topic) is not among the mandatory study subjects in high school in Finland, it is not part of the matriculation exam [3]. Therefore, the entrance exam to CS/IT does not contain programming per se; instead it contains logical problems and essay writing. Many of the students who are admitted to computer science do not have an accurate image of the subject, and many drop out soon after their studies have started.

Another issue worth noting is that there are no tuition fees for anyone in Finland (from elementary schools to universities). Instead, the government supports students by a monthly allowance for living expenses, including rent support. As CS/IT is not among the most highly sought-out study subject, some students apply for CS/IT as a fallback position, and accept the study right in order to get the student benefits. Instead of studying CS/IT, they use the extra year for e.g. preparing for an entrance exams to a more preferable study subject.

In order to alleviate the problem of having an incorrect mental image of CS/IT studies, we wanted to allow high-school students (esp. grades 10 to 12) in Finland to experience CS/IT studies. Therefore, we opened up our introductory programming course (CS1) to the whole country[4], targeting especially high-school students who have no programming education in their schools, or who seek more advanced courses than their local high school offers[5].

The most significant benefit is that the MOOC participants get a more realistic view of the studies they would be encountering if they took CS/IT as a major subject, and can themselves evaluate if they are up to it. By completing the MOOC in programming, the students show us at the department that they are both competent and persevering enough to study CS/IT. Therefore, it is only natural to grant those students full study rights for a degree.

3. MOOC AS AN ENTRANCE EXAM

Starting the MOOC is straightforward, as there is no need to provide any other information than a valid email address when registering. If the student seeks admission, she is required to enter full personal information. The option for applying for the study right is available for the first two months.

3.1 Course Content and Pedagogy

The MOOC in programming is content-wise exactly the same as our CS1, which is taught in Java using an objects-early approach. The course contains 12 weekly exercise sets, and covers topics typical to any introductory programming course; assignment, expressions, terminal input and output, basic control structures, classes, objects, methods, arrays and strings, advanced object oriented features such as inheritance, interfaces and polymorphism, and familiarizes students with the most essential features of Java API, exceptions and file I/O[6].

During the MOOC, the participants work on over 150 programming exercises, which are further split into over 350 tasks. The students that are applying for the study rights must correctly solve 80% or more of the weekly tasks in order to be invited to the interview. The material is handed out online in a book-like format, with a few screencasts, and its sole purpose is to help the students work on the exercises; the main working method for the students is programming.

The learning-by-doing orientation comes from using the Extreme Apprenticeship (XA) [17] method in the course implementation. XA is based on cognitive apprenticeship [3, 2] and approaches programming as a craft that needs to be

[3]Many schools offer computing as an elective course. However, as there is no national curriculum for courses in computing, courses often concentrate on the use of computer applications and computer literacy. The situation in Finland as such resembles many other countries, e.g. USA [20].

[4]As is typical for MOOCs, there are no restrictions for participation. Our MOOC is in Finnish language, so the natural audience is mostly in Finland.

[5]In case a participant does not want to apply for a study right but would like to have a certificate of accomplishment, we have facilitated the schools in Finland to provide examinations. High schools use the certificate for granting school credits.

[6]The course material and exercises are available at http://mooc.fi and licensed under the Creative Commons BY-NC-SA -license.

honed continuously. Two core values in XA are "practice as long as necessary" and "continuous feedback". In our earlier XA-based courses [11], the feedback has been provided by human advisors (teachers). In the MOOC, the participants program in an industry-standard programming environment that contains a plugin, which provides help for the students (for additional details, see [18]).

3.2 Interview and Programming Task

Once the students have worked through the required number of programming tasks, they are invited to an interview. The interview is a two-part process: first, the students work on a programming task in a live setting, and after that are interviewed by two members of the faculty. The programming task is done in a lab, where a supervisor can help participants with e.g. operating system or programming environment-related issues, and correct potential misunderstandings regarding the task. The students are free to use any available material which can be found on the internet, e.g. the course material. However, asking for help in solving the programming task is not allowed.

The participants had a total of 2 hours for the task, which was as follows for the interview held during spring 2012.

Programming task: a text analyzer

Create an application that can be used to analyze text file contents. The application should contain at least the following features:

- calculating the number of words in a file
- finding and printing the most common word(s) in a file
- finding and printing the longest word in a file

If you wish, you can also create additional features.

The program should be able to analyze several files during a single execution, and it should also be able to handle large files. You can test your application for example with Kalevala, which is available at
http://www.gutenberg.org/cache/epub/7000/pg7000.txt

You can decide what sort of a user interface the application provides, however, we suggest that you build a text-based user interface. Below is an example of how the application could work:

```
Enter filename, empty input exits the program
> kalevala.txt
commands: longest, words, most-common, help
command > words
67443
command > longest
longest word is: kautokengän-kannoillansa
command > most-common
most common word is: on
>
finished processing kalevala.txt

Enter filename, empty input exits the program
> test.txt
commands: longest, words, most-common, help
command > help
commands: longest, words, most-common, help
command > words
7
command >
finished processing test.txt

Enter filename, empty input exits the program
>
Thank you!
```

After the programming task, the students are interviewed for up to 30 minutes by two faculty members. The faculty members discuss the students' program design choices and possible issues with e.g. performance with the student. During the interview, the faculty also attempts to form an understanding of the student's background, and reasons for applying to the department of computer science. Things that are of interest are e.g. existing programming background, the student's vision regarding her life after five years from now, and existing educational background.

3.3 Selection of Students

During spring 2012, most of the students that did over 80% of the exercises in the MOOC in programming and applied for study rights also fared well in the actual interview. Most of the participants were able to complete the programming task fully, and only a handful of the participants had issues with e.g. program design or did not have a working program at all. Out of the 52 students that applied for a study position during spring 2012, 49 study rights were granted. Out of the 49, 38 students started their studies during fall 2012, and the remaining 11 had varying reasons not to start their studies: they are still in high school, they postponed the start due to the mandatory military service, or they took another, preferred study position.

The number of applicants via the traditional path has been in hundreds for years. Therefore, we are not expecting an uncontrollable need to scale up the interview process. Currently, the interviews involved with the MOOC entrance have been conducted by two faculty members without extra resources.

4. DATA

Our data contains study records from students that have started their studies at the Department of Computer Science at the University of Helsinki in August 2012. As some of the students postpone their start due to the military service, focus on other studies than Computer Science, or have transferred courses from earlier studies (e.g. open university), we include only students that have either attempted or completed the introductory programming course during the academic year 2012-2013.

The study records cover the period from August 1, 2012 to May 24, 2013. We examine two separate groups. The first group (MOOC, n=38) contains students that have been admitted via the programming MOOC that was organized during spring 2012. The second group (NORM, n=68) contains students that were admitted via the traditional path, namely the entrance exam, matriculation exam, or a combination of both. The MOOC group has the introductory programming

CS/IT Courses		
	MOOC (n=38)	NORM (n=68)
Credits		
overall	1257	1629
mean	33.08	23.96
std	11.32	15.3
median	32.5	24
Courses passed		
overall	346	452
mean	9.11	6.65
std	2.95	4.02
median	9	7
Courses failed		
overall	66	157
mean	1.74	2.31
std	1.83	2.37
median	1	2
Grade stats		
mean	4.04	3.79
std	1.15	1.2
median	4	4

Table 1: Student performance in CS/IT-related courses.

Math Courses		
	MOOC (n=38)	NORM (n=68)
Credits		
overall	273	350
mean	7.18	5.15
std	7.49	7.63
median	5	0
Courses passed		
overall	46	62
mean	1.12	0.91
std	1.19	1.25
median	1	0
Courses failed		
overall	30	50
mean	0.79	0.74
std	0.74	0.66
median	1	1
Grade stats		
mean	3.39	3.37
std	1.2	1.45
median	4	4

Table 2: Student performance in mathematics courses.

and advanced programming courses (a total of 9 ECTS[7]) included in the data, as the courses have been added to students' records when they were granted study rights, i.e. 1st of August 2012. We offered the MOOC for all students that were admitted as well. The second group (NORM) does not include students, who took the MOOC during the summer (n=15), as their effective study time would be 3 months longer than the other students in NORM group, causing additional deviation in the data.

For each study subject (CS/IT, Math, all), we report the number of credits, number of courses passed, number of courses failed, and grades for each category. The grades range from 1 (pass) to 5 (excellent), and the grade averages exclude failed courses. Our university does not force courses to be graded on a bell curve. On the contrary, student grades are based on the true performance of the student using an explicit criteria.

When looking at the data, one should keep in mind that the study path for first-year students is designed for students taking the programming courses during the first semester. This means that the students that have been admitted via MOOC have received no benefits from a tailored study path.

4.1 CS/IT Courses

Table 1 contains the students' performance in CS/IT courses. When considering the number of credits that students have gathered during the study period, the average is almost identical when we include the knowledge that MOOC students have taken the introductory programming courses earlier. The standard deviation in the number of credits, which is higher for the NORM group, indicates that there is more variance within the NORM group. In essence, it indicates that there are students that end up failing their first programming courses and do not proceed at all, as well as students, who fare well in their studies.

On average, the students admitted via a MOOC pass more CS/IT courses than the NORM group, and end up failing less courses. On average, MOOC students have one fail per five passed courses, while the NORM group has one fail per three passed courses. The standard deviation in both passed and failed courses is also smaller for the MOOC group; on average, the MOOC students fare better than the NORM students. This is also seen in the grade statistics; although there is not much difference, and the median grade is 4 on a scale from 1 (pass) to 5 (best) for both groups, the average grade is slightly higher for the MOOC group.

4.2 Mathematics Courses

In Table 2, we see the students' performance in mathematics courses. Although mathematics is not a mandatory minor subject, completing at least 10 ECTS of mathematics is mandatory. Typically, students enroll in a course called Introduction to University Mathematics, which covers the essential mathematics required for the course on Data Structures (CS2), where e.g. algorithm run-time analysis is one of the focus areas.

On average, both student groups have completed at least 5 ECTS of mathematics during their first year of studies. The MOOC students have taken over 7 ECTS worth of mathematics, while NORM students have 5.15 ECTS. Note, however, that the standard deviation is high for both groups, which means that it is very likely that there are students in both groups that have either not passed any mathematics courses, or have passed more than one mathematics course.

When looking at the number of passed courses, the median for the MOOC students is 1, and the median for NORM students is 0. This means that one half or more of the NORM students have not succeeded in passing any mathematics courses. This is problematic, as although mathematics is

[7]European Credit Transfer and Accumulation System. An academic year corresponds to 60 ECTS, and one ECTS credit point equals 25-30 hours of student work.

	All Courses	
	MOOC (n=38)	NORM (n=68)
Credits		
overall	1675	2296
mean	44.08	33.76
std	17.58	21.96
median	43	32.5
Courses passed		
overall	434	599
mean	11.42	8.81
std	4.24	5.27
median	11	9
Courses failed		
overall	101	214
mean	2.66	3.15
std	2.16	2.73
median	2	3
Grade stats		
mean	3.94	3.73
std	1.13	1.18
median	4	4

Table 3: Student performance in all courses.

not a formal requirement for CS2, it is highly beneficial for students to understand the contents of the Introduction to Mathematics course as they take on Data Structures.

The grade averages for both groups are almost alike; the only difference being the slightly higher standard deviation for the NORM group.

4.3 All Courses

Table 3 contains information on all the courses that the students have taken during their first year of studies. It contains both the CS/IT courses and the mathematics courses, and in addition other courses that the students may have taken. Students are able to choose almost any course from any discipline, so minor studies vary a lot among the students. Among the students that have started their studies in 2012, we have students taking courses related to e.g. politics, economics, literature, psychology, languages and law.

Overall, the students in the MOOC group fare slightly better on average than the NORM group, but the NORM group has more variation. On average, the MOOC students have gathered 44.08 ECTS during their first year (35.08 if programming courses are not included), while the NORM students have gathered 33.76 ECTS. There is a small, but noticeable difference, and the median is 43 for MOOC (34 if programming courses are not included), and 32.5 for NORM[8].

When looking at the number of courses passed, and the number of courses failed, the MOOC students fare better on average, while the NORM students have a larger variation. The MOOC students have one fail per four passed courses, while the NORM students have one fail for slightly less than three courses. Again, some students perform well, while others perform poorly. The grade statistics are almost alike, on average the grade of MOOC students is 3.91, while the grade average for NORM students is 3.71.

In addition, when considering the amount of students that have received less than 10 ECTS during their first two semesters, i.e. have done only the programming course or less, only one out of the 38 MOOC students did not complete anything outside the programming courses. When considering the students in the NORM group, a total of 12 students (17.6%) have gathered less than 10 ECTS. We must note that we consider only the students that started their studies and participated in the introductory programming course; in reality, the number is higher.

[8]It should be noted that the student should complete 60 ECTS per academic year in order to graduate according to the model curriculum. In practise, a slow start and advancement of CS/IT studies (as well as large dropout rate) is a common problem in Finland. Even the most competent students tend to start working in the IT industry while studying, thus delaying their graduation.

5. DISCUSSION AND FUTURE WORK

Our initial analysis of students that have been admitted via the MOOC indicates that they are failing less courses and gaining slightly more credits than the students admitted via the traditional path. However, lots of variance in the student groups exist, and both of the groups have so-called high performers and low performers. As we compared the MOOC students to students that have attempted or succeeded in the introductory programming courses during the academic year 2012-2013, our initial analysis excluded the admitted students that did not study at all (e.g. entered military service or started to study another subject at the university) or chose to start their studies early by participating in a voluntary MOOC during summer 2012.

At the University of Helsinki, Department of Computer Science, we receive some 500-600 study applications per year. A majority of the applicants seek a study right via the entrance exam, while some apply directly using their matriculation exam score. Typically less than 200 students are admitted, and of these, on average, less than 130 students accept the study right. Thus, CS/IT is not the number one choice for the study for many of the applicants. Moreover, some 20-30 students do not start any CS/IT courses, even if they accept the study right. When we compare these traditional figures to our first MOOC intake, in which over 93% of the applicants were accepted and started their studies accordingly, the MOOC intake is far superior in matching the students to an appropriate and desired area of study.

Having the students successfully perform introductory programming courses already before they start their studies gives the students a head start over their fellow students. It also acts as a preliminary verification on the students' motivation to study CS/IT. In addition, the students are not getting stuck to the "filter" of learning to program that is a cause for challenges for many in their early studies.

As the awareness of our MOOC as an entrance exam is increasing, we are currently in the process of increasing the number of students admitted via the MOOC. In spring 2013, a total of 66 students were admitted. In addition to improving the intake, we are also working on the students' first year experiences so that the MOOC students have more relevant courses to work on. Even though our MOOC has proven to be beneficial for us, we are not aiming to stack up on online education: we want all of our students to participate in the academic community and therefore emphasize the social support during the degree studies, helping them in the transition from a high school to the university [19].

We see a strong indication that one of the important success factors in first-year CS/IT studies is foundational programming skills. These skills can be practised already before the formal start of the degree studies. Universities with a similar admission system to ours that are facing challenges with student intake and performance (e.g. students dropping out during first semester, students not opting for CS/IT-studies) may benefit from a long-term programming exam, which is administered already during the high-school studies (cf. e.g. [4, 9]).

Acknowledgements

This research is partially funded by the Technological Industries of Finland Centennial Foundation. We gratefully acknowledge the anonymous reviewers for their valuable feedback.

6. REFERENCES

[1] M. E. Caspersen, K. D. Larsen, and J. Bennedsen. Mental models and programming aptitude. In *ACM SIGCSE Bulletin*, volume 39, pages 206–210. ACM, 2007.

[2] A. Collins, J. Brown, and A. Holum. Cognitive apprenticeship: Making thinking visible. *American Educator*, 15(3):6–46, 1991.

[3] A. Collins, J. Brown, and S. Newman. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In *Knowing learning and instruction Essays in honor of Robert Glaser*, volume Knowing, l of *Psychology of Education and Instruction Series*, pages 453–494. Lawrence Erlbaum Associates, 1989.

[4] T. Crick and S. Sentance. Computing at school: stimulating computing education in the UK. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 122–123, New York, NY, USA, 2011. ACM.

[5] J. Daniel. Making sense of MOOCs: Musings in a maze of myth, paradox and possibility. 2012. http://www.academicpartnerships.com/docs/default-document-library/moocs.pdf.

[6] K. Devlin. The future of textbook publishing is us, 2012. http://devlinsangle.blogspot.fi/2012/08/the-future-of-textbook-publishing-is-us.html.

[7] S. Downes. What is a connectivist MOOC. 2012. http://www.connectivistmoocs.org/what-is-a-connectivist-mooc/.

[8] G. E. Evans and M. G. Simkin. What best predicts computer proficiency? *Commun. ACM*, 32(11):1322–1327, Nov. 1989.

[9] B. Franke, J. Century, M. Lach, C. Wilson, M. Guzdial, G. Chapman, and O. Astrachan. Expanding access to k-12 computer science education: research on the landscape of computer science professional development. In *Proceeding of the 44th ACM technical symposium on Computer science education*, SIGCSE '13, pages 541–542, New York, NY, USA, 2013. ACM.

[10] P. Kinnunen, R. McCartney, L. Murphy, and L. Thomas. Through the eyes of instructors: a phenomenographic investigation of student success. In *Proceedings of the third international workshop on Computing education research*, ICER '07, pages 61–72, New York, NY, USA, 2007. ACM.

[11] J. Kurhila and A. Vihavainen. Management, structures and tools to scale up personal advising in large programming courses. In *Proceedings of the 2011 conference on Information technology education*, SIGITE '11, pages 3–8. ACM, 2011.

[12] A. McAuley, B. Stewart, G. Siemens, and D. Cormier. The MOOC model for digital practice. 2010. http://davecormier.com/edblog/wp-content/uploads/MOOC_Final.pdf.

[13] MOOCs@Edinburgh Group. MOOCs @ Edinburgh 2013: Report nr. 1, 2013. http://hdl.handle.net/1842/6683.

[14] G. Siemens. What is the theory that underpins our MOOCs? 2012. http://www.elearnspace.org/blog/2012/06/03/what-is-the-theory-that-underpins-our-moocs/.

[15] Simon, S. Fincher, A. Robins, B. Baker, I. Box, Q. Cutts, M. de Raadt, P. Haden, J. Hamer, M. Hamilton, R. Lister, M. Petre, K. Sutton, D. Tolhurst, and J. Tutty. Predictors of success in a first programming course. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 189–196, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[16] A. Vihavainen, M. Luukkainen, and J. Kurhila. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th annual conference on Information technology education*, SIGITE '12, pages 171–176. ACM, 2012.

[17] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 93–98. ACM, 2011.

[18] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using Test My Code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, ITiCSE '13, pages 117–122, New York, NY, USA, 2013. ACM.

[19] P. Wilcox, S. Winn, and M. Fyvie-Gauld. 'It was nothing to do with the university, it was just the people': the role of social support in the first-year experience of higher education. *Studies in higher education*, 30(6):707–722, 2005.

[20] C. Wilson, L. A. Sudol, C. Stephenson, and M. Stehlik. Running on empty: The failure to teach k-12 computer science in the digital age. Association for Computing Machinery. 2010.

Publication III.1

Arto Vihavainen, Juha Helminen, and Petri Ihantola

How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces

In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research (Koli Calling '14)*

How Novices Tackle Their First Lines of Code in an IDE:

Analysis of Programming Session Traces

Arto Vihavainen
Department of Computer Science
University of Helsinki
Finland
avihavai@cs.helsinki.fi

Juha Helminen
Department of Computer Science and Engineering
Aalto University
Finland
juha.helminen@aalto.fi

Petri Ihantola
Department of Pervasive Computing
Tampere University of Technology
Finland
petri.ihantola@tut.fi

ABSTRACT

While computing educators have put plenty of effort into researching and developing programming environments that make it easier for students to create their first programs, these tools often have only little resemblance with the tools used in the industry. We report on a study, where students with no previous programming experience started to program directly using an industry strength programming environment. The programming environment was augmented with logging capability that recorded every keystroke and event within the system, which provided a view on how the novices tackle their first lines of code. Our results show that while at first, the students struggle with syntax – as is typical with learning a new language – no evidence can be found that suggests that learning to use the programming environment is hard. In a two-week period, the students learned to use the basic features of the programming environment such as specific shortcuts. Although we observed students using copy-paste-programming relatively often, most of the pasted code is from their own previous work. Finally, when considering the compilation errors and error distributions, we hypothesize that the errors are a product of three factors; the exercises, the environment, and the data logging granularity.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education

Keywords

introductory programming; source code snapshot analysis; programming session trace analysis; programming behavior; data mining; learning analytics; novice programmers; programming environments; novices and programming environments

1. INTRODUCTION

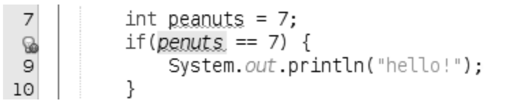
One of the factors that defines whether students choose to pursue a career in a field is their success in the introductory courses [4]. If students fail the very first introductory courses, many lack the belief that they could succeed in that area in the future. This effect is visible also in computing, and thus, as programming courses are among the first courses that students attend in computing, it is not surprising that introductory programming courses have received plenty of attention from computing education researchers [20].

These days, programming instructors often include a number of activities in their courses, including opportunities to read code, study worked examples [17, 18], and to use program visualization tools [25]. However, as one of the core learning goals of an introductory programming course is to learn the art of writing programs, the instruction eventually will include more and more activities in which the students write programs themselves [1]. Here, novice-friendly programming environments such as Alice, BlueJ or Scratch [5, 16, 23], or depending on the course, online programming environments that are embedded into the learning material management come into play (see e.g. [19, 22]).

Based on recent studies that report that, overall, there has been little improvement in the pass rates of university-level introductory programming courses [31, 28], we are faced with the question that are we taking the correct turn when embedding our courses with *systems that make the first steps easier*? Could it be that we are blinded by the recent push towards making programming accessible to everyone, and as a consequence, forgetting that industry-strength programming tools are constantly improved, and used by millions around the world? Are these programming environments truly so hard for novices to use that easier ones need to be built?

While a growing body of research exists on the struggles that students face when writing code within educational environments (see e.g. [2, 3, 10]), little research exists on the issues that students face when learning to program within standard, off-the-shelf programming environments. This is somewhat surprising, as one of the implicit reasons for the creation of these educational environments is that *the standard off-the-shelf programming environments are complex and hard to use.* This suggestion implicitly dismisses the features that many modern programming environments boast with, such as the continuous compilation of source code that creates up-to-date error information (see Figure 1), source code auto completion, various debugging features, and the overall support for numerous programming languages.

While these features are all generally available in free and open-source editors such as Eclipse [1] and NetBeans [2], educational programming environments typically lack them. Undoubtedly, these kinds of features may have a significant impact on how novice programmers work and learn.



```
int peanuts = 7;
if(penuts == 7) {
    System.out.println("hello!");
}
```

Figure 1: NetBeans IDE automatically identifies syntax errors, highlights them, and provides suggestions on how to fix them (suggestions visible on mouse click).

In this study, we seek to understand how novice programmers tackle their first lines of code when programming is started directly using an IDE. We investigate what are the things that novice programmers struggle with and study how their behavior changes over a short period of time. The participants in the study have stated that they have no previous programming experience, and have been exposed to programming directly within the NetBeans environment, without being first exposed to e.g. educational programming environments. The data that our analysis is based on is fine-grained, where every key-press within the programming environment has been recorded. This means that no assumptions on the programming behavior between subsequent snapshots need to be taken.

2. RELATED WORK

While there are no studies similar to ours in the scope and data granularity, a notable amount of research that analyzes how students' proceed in their first programming courses exist. Many of these studies, unfortunately, are unclear on the details regarding students' programming experience, and the term *novice* is often defined loosely or not at all. In most cases, all students that attend an introductory programming course are considered to be novice students, while in reality, many may have previous programming experience, and e.g. attend the course due to degree requirements or in the hope of easy credits.

During the last decade, tools that make it possible to gather data from students' programming environments have slowly become mainstream, and the analysis of snapshots is at the point where tools designed for snapshot analysis are starting to pop up (see e.g. [9]). However, many of these are oriented towards coarser data, and little work on fine-grained data analysis exists. Here, we review relevant studies where snapshot data of coarser granularity has been utilized for multiple purposes.

Perhaps the closest study to ours is the one by Heinonen et al. [9]. While the main focus of their work is on demonstrating a tool for source code analysis, they also analyzed snapshots from students that had no existing programming experience and had either passed or failed an introductory programming course. However, their analysis focused on style-related coding errors and reasoning issues, they had a small dataset, and the data was coarse grained, containing only students' saves, code executions, and compilations. Another work that analyses students' behavior within a programming environment is the work by Dyke [7], where the analysis focused on students' use of IDE features. He found that the use of features such as code generation had a high correlation with course success, but, did not discuss the students' backgrounds. Thus, it is possible that the results could be partially explained by past experience.

One notable research direction in snapshot analysis is data-driven approaches for identifying at-risk students. In these approaches, the programming environment is instrumented to gather snapshots based on specific triggers such as time interval or actions that the student takes within the environment, e.g. saving source code or running the program. Jadud was perhaps the first one to propose an approach, where students' subsequent compilation events were analyzed, and the persistence of different compilation errors was quantified to produce an *error quotient* [14] that correlated with success. Jadud's work was later extended by Watson et al. [30], who took the relative performance differences of the student population into account, leading to an improvement in the predictive power.

Others have analyzed the time usage of students' within a programming course and the use of programming environment features. For example, Edwards et al. investigated the study behavior of novice programmers, and found that students who received both high and low marks from programming assignments typically received the high marks from work that they started and ended well before deadline, while the low marks came typically from assignments that the same students started relatively close to the deadline [8]. Similar results were later identified by Vihavainen, who identified students' eagerness (or earliness) to start working on the programming exercises as one of the factors that distinguished students into high-performing, low-performing, and failing students [27].

The errors themselves have also been analyzed. While Jadud outlined the most typical errors that students' see when programming in BlueJ [13], Denny et al. outlined that some errors are harder to fix than others, and that the errors may be related to the programming environment [6].

Another research direction is modeling students' progress for e.g. the use of a tutoring system or for feedback generation. Here, Piech et al. [21] modeled students' programming patterns within an assignment where students programmed using Karel the Robot system. They found that the usage patterns were more predictive of midterm scores than the scores from the assignment. While students' progress within programming courses are generally considered harder to model than the progress within an individual assignment or a micro-world, some notable attempts have been exercised by Hosseini et al., who explored programming concept reduction and increment patterns within a programming course [11], and experiments by Yudelson et al., who explored the use of knowledge models used to track students' understanding of course concepts in a Java MOOC [32].

Data gathered from snapshot systems has also been used to detect emotions and intentions. For example, Vee et al. [26] did a preliminary study where they sought to detect

[1]https://www.eclipse.org/
[2]https://netbeans.org/

intentions of students from snapshot data, while Rodrigo and Baker sought to identify frustrated students using both logged data and notes from external observers [24]. For further information on platforms doing fine grained logging and related research in this field, see e.g. Helminen et al. [10].

3. METHODOLOGY

To provide gain insight on how novices write their first lines of code, this research answers the following questions:

- How do novice programmers write their first lines of code?
- What are the most typical syntax errors that novice students encounter? Are they any different to those reported in literature related to educational programming environments?

We define novice as a student who has *explicitly stated that he or she has no previous programming experience*, and use NetBeans as the programming environment. Data from the students' progress is gathered by a NetBeans plugin called Test My Code [29], which we have altered to gather finer grained data. The novice programmers and assignments under study are from an introductory programming course in Java that was offered by the University of Helsinki during Spring 2014. The course was open and free for anyone willing to participate (i.e. no need to be enrolled at an educational institution). To focus on the struggles of novices, we utilize only two first weeks of the course, and limit our study to those students that indicated in a voluntary survey that they have no past programming experience (n=233, average age=23.8, σ=5).

The course starts as many other programming courses, i.e. by programming a "Hello World"-application. Over the two weeks, the students work on 42 exercises, some of which are split into smaller tasks, and practice the use of input, output, conditional statements, loops and methods. All exercises are worked in the NetBeans IDE with Test My Code that authenticates students and records data from each keypress. That is, for each keypress (or button click) within the IDE, information on student, source code change, time, and the exercise that the student is working on is stored. The source code changes are recorded as diffs between keystrokes that are augmented with additional descriptors related to whether code has been inserted, removed, or pasted (pasting is identified by non-empty strings with length more than one with the system clipboard). Undos and redos are considered as inserts and removes, depending on the change.

To answer the research question "*How do novice programmers write their first lines of code?*", two researchers performed an in-depth qualitative analysis on two programming exercises where students perform printing, do comparison and use loops.

The research question "*What are the most frequent syntax errors that novice students encounter? Are they any different to those reported in literature related to educational programming environments?*" is answered by performing a quantitative analysis on the events gathered. We analyze the events on multiple granularities, and contrast our results to those previously reported in the literature.

4. RESULTS

In the next section, we present our findings from the qualitative analysis of how print statements are constructed and a quantitative look into how copy and pasting is used by the students. Then, the results of the quantitative analysis of compilation statistics and errors will be presented.

4.1 Writing First Lines of Code

Qualitative Overview of How Students Write Print Statements

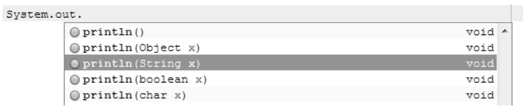
To get an overview of how students write their very first lines of code – print statements – we manually examined the key-level logs from two early assignments. We performed content analysis on the very first assignment on the course and another from the second week. Both exercises required students to print something. We focused on how the print statements (i.e. `System.out.print("something");` or `System.out.print("something");`) are typed. After viewing the traces one-by-one and recording the approaches, the following categories of approaches for typing the print statement emerged:

linear In the *linear* approach, students type the print command `System.out.println` character by character from left to right. In most cases, this is followed by the IDE automatically adding parentheses and a semicolon at the end of the method name. Typos and other errors may occur as later discussed.

autocomplete Most IDEs support autocompletion. In addition to adding parentheses at the end of method calls, NetBeans suggests completions for classes and objects. For example, typing `System.` will open a popup with alternative completions as illustrated in Figure 2. Typing more characters will limit the completions in the popup accordingly.

sout As the print statement is commonly used but rather long in Java, NetBeans offers a shorthand `sout` that, when typed and followed by pressing tab, is automatically replaced with `System.out.println("");`. In addition, the cursor jumps inside the quotes so that the user can immediately go on to type the contents of the string. The course material hints the students to try the command after they have worked on a number of programming exercises.

copy-paste The last observed category of typing the print statement is to copy and paste code from somewhere else and then edit the argument only.



```
System.out.
        println()               void
        println(Object x)       void
        println(String x)       void
        println(boolean x)      void
        println(char x)         void
```

Figure 2: NetBeans IDE suggests alternative completions upon entering the dot operator.

Table 1 summarizes how frequently the different approaches were observed in the two assignments selected. The approaches overlap and it is quite common that different approaches are used even during a single session. In the statistics, the linear category contains only the cases where this approach has been used without any autocompletion. We observe that in the very beginning students are using copy and paste a lot whereas already on the second week `sout` shortcut is the most common approach. At the same time, problems related to typing the print statement have almost disappeared. During the first week, problems included first trying to use the `sout` shorthand and failing – maybe they, for example, forgot the key used to invoke autocompletion. This was followed by erasing the `sout` and starting to type the print statement linearly. Many who did this had some problems also with their second approach. For example, 1) incorrectly mixing capital and small letter (e.g. `system` instead of `System` and `Out` instead of `out`), 2) using other characters than the dot to access class members (e.g. `System-out-println`), 3) various problems related to defining string literals, for example not using quotes at all (i.e. `System.out.println(Hello);`) or using parentheses instead of quotes (i.e. `System.out.println((Hello));`), and forgetting `.out` from the `System.out.println` and 4) normal typos without obvious misunderstanding behind.

Copy-paste programming

In analyzing how learners formed print statements, we observed quite a few using a copy-paste approach in the second assignment during the first week but by the second week the learners had almost unanimously moved on to making use of the `sout` shorthand. In order to study the general popularity of a copy-paste approach, we performed an additional quantitative analysis of how frequently this occurs during the first two weeks of the course. This includes in total 42 assignments.

The fine-grained key-level trace allows us to identify the moments when several characters appear in the code at once in a single event. There are essentially two cases when this happens. Either the learner has made use of some special IDE code generation feature such as autocompletion or the learner pastes code from the clipboard. Currently, the actual pasting functionality in the IDE is not instrumented. Instead, we classify the cases where added character sequences are longer than a single character and match the current contents of the user's clipboard as pastes. Thus, for example, we are not including, nor would we currently be able to, any occurrences of pasting a single character.

In terms of our identification method, overall, 210 learners copied and pasted something into their code in the assignments of the first two weeks. In total, learners pasted text 20351 times in the 42 assignments examined. We further analyzed the pasted text and tried to infer its origin. One likely source for copied code is the learner's own previous work on the course in a similar assignment. In addition to checking whether the code could have been copied from the learner's current code, assuming that the learners more or less do the exercises in order, we can look back and try to match copied code to anything the learner wrote earlier. This way we get an approximation of the amount of copying the learner is likely doing from his or her own work or from that given in the code templates of previous assignments. In total 13384 paste events, that is, around 66% of them, could be placed in this category leaving out about a third of paste events whose text probably originated elsewhere in the learning material or on the web.

week	n	all	5 sec	10 sec	snapshot	submit
1	233	44.7%	61.9%	65.5%	90.6%	99.4%
2	179	36.4%	60.5%	65.3%	94.5%	99.6%

Table 2: Percentage of the states in each granularity that compile successfully for the first two weeks.

There is notable variance in the amount of copy and pasting being performed both between different exercises and between learners. In assignment 13, the total number of paste events was as high as 5034 while in a few assignments there were less than 60 paste events across all learners. Here, the behavior is explained by the assignment type – the assignment included creating repetitive actions for a robot, and students had not yet been taught loops.

The number of pastes done by each learner in the 42 assignments ranged from less than 10 to over 300.

4.2 Compilation Statistics and Errors

Compilation Statistics

Occurrence of compilation errors and the error message types are used to identify at-risk students [15, 30]. When students work in programming environments such as BlueJ, it is typical that only some 50% of the snapshots that are taken when students' save, compile or run the application compile. Here, we look at the overall compilation statistics on multiple granularities over the two weeks. The granularities are defined as follows:

- *all* - every key-stroke and event in the data set is compiled: this contains e.g. states where students' are in the middle of typing something
- *5 seconds* - if students pause for 5 seconds, the data is stored: this indicates e.g. a short pause in typing or a thinking break
- *10 seconds* - if students pause for 10 seconds, the data is stored: this indicates a bit longer pause in typing or a thinking break
- *snapshot* - states where the student has saved, run, debugged or tested the application
- *submit* - states where the student has used the submission feature of the Test My Code-plugin (i.e. sending the solution to an assessment server)

Table 2 shows the average (mean) percentage of states that compile. Over 90% of submissions and snapshots compile during both weeks, and over 60% of the states that are stored when a student takes a pause from programming compile. Furthermore, over 35% of the key-stroke states compile during both weeks. When comparing these numbers to the numbers reported with e.g. BlueJ, there is a considerable difference.

Figure 3 shows the distribution of compilation errors for the different granularity types over the two week period. Both snapshots and submissions compile in majority of the cases, while the key-strokes compile less frequently.

		N	linear	auto-complete	sout	copy-paste
Assignment 2	(week 1)	121/36	27/18	33/15	72/9	22/4
Assignment 27	(week 2)	122/1	6/-	2/1	112/-	4/-

Table 1: Different approaches to write a `System.out.println()` statement / how many students using that approach had to edit the line by deleting something in two exercises from the first and second week of the course. N is the number of students who have typed print statements in the solutions.

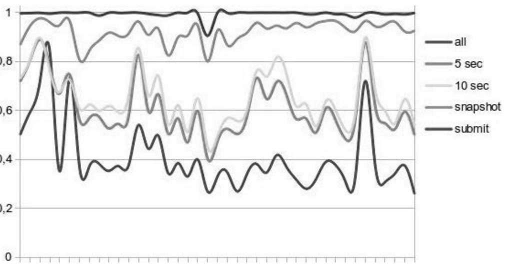


Figure 3: Compilation success percentages over all exercises with four different granularities. The topmost line indicates the compilation successes for submission, the next row snapshots, then keylevel-events with 10 second window, then 5 second window, and finally all keystrokes.

Compilation Errors

Compilation errors and how students tackle them are the cornerstone of many algorithms that identify at-risk students. Here, we consider the compilation errors that students face within the NetBeans environment, and compare those to those reported in the literature. We consider the articles from Jadud [13] and Denny et al. [6] as the comparison points. In Jadud's work "the five most common errors account for 58% of all errors generated by students while programming: missing semicolons (18%), unknown symbol : variable (12%), bracket expected (12%), illegal start of expression (9%), and unknown symbol : class (7%)" [13], and in the work of Denny et al., the five most common errors account for 72.3% of all errors; cannot resolve identifier (24.0%), type mismatch (18.4%), Missing ';' (13.0%), token should be deleted (10.3%), and method not returning correct type (6.6%).

In our context, when considering the compilation errors that exist in the snapshots after the student has taken a five second pause, the five most common errors account in week one account for 77.4% of all compilation errors in week 1 and 80.3% of the errors in week 2, as shown in Table 3. On the other hand, if we consider snapshots that are taken when students save, run or test their code (see Table 4), during the week one, the top 5 errors account for 62.1% of all errors, while the top five errors account for 65.8% during the week two.

As Denny et al. note, comparing the error outputs is challenging as the error messages vary across tools. For example, the standard Java error message codes, e.g. "compiler.err.expected3", do not always distinguish specifics, and in the previous case, one cannot determine whether the missing, or unknown, symbol is a variable or a class from the message code. In our data, when considering both the snapshots and the data gathered after a 5 second pause, less than 15% of the errors are related to not being able to find a symbol; on the other hand, these numbers are slightly less than 20% in Jadud's data, when variable and symbol errors are aggregated. Similarly in our context, the errors indicating a missing identifier or a semicolon are aggregated based on the code, which makes the comparison difficult.

error	week 1	week 2
<identifier >, ';' or ')' expected	30.0%	29.0%
illegal start of expression	15.3%	19.3%
not a statement	12.1%	14.0%
premature end of file	14.1%	9.5%
cannot find symbol	5.9%	8.5%

Table 3: Top 5 compilation errors for weeks one and two when student has taken a five second pause from programming.

error	week 1	week 2
<identifier >, ';' or ')' expected	26.9%	23.3%
premature end of file	10.5%	9.0%
cannot find symbol	8.2%	13.9%
illegal start of expression	8.7%	8.9%
class, interface, or enum expected	7.8%	10.7%
not a statement	7.4%	5.7%

Table 4: Top 5 compilation errors for weeks one and two when student based on snapshots that are recorded on student save, run and compile.

5. DISCUSSION

We have investigated the programming sessions of complete novices (as reported by themselves). In the first assignments, they compose their first programs starting with a simple class template with a single method and adding only print statements of the form `System.out.println();`. At this point, the statement is more or less a sequence of characters that has no understandable structure to them. They simply try to copy it from the learning material character by character but end up, for example, mixing capital letters, forgetting parts of it, and changing the order because to them there may not seem to be any obvious logic. Not only this, but surrounding the space where they are to fill in the print statements there is more code that they simply cannot comprehend at this point. In our analysis of how learners form some of their first print statements we did find them struggling. Special teaching languages or languages such as Python that have an arguably cleaner syntax than Java could definitely have their benefits.

One of the more peculiar mistakes learners made, was subtituting the `l` in the print statement with an `I`. In a way, this makes sense – printing `In` something is more intuitive than printing an `ln` – a line – unless explained in the learning material. The letters also resemble each other quite a lot in some fonts. Truly, even such less apparent issues in the usability of programming material and environments may have adverse effects in learning.

In regard to our further analysis of the print statement, it is also interesting to note, that while it seems that already on the second week the learners were not struggling with the statement, neither were they actually writing it anymore. Instead, almost all of them used the `sout` shorthand provided by NetBeans. Whether they could type a print-command out of memory without slips after two weeks of practice remains an open question.

Looking at our findings about copy and pasting, we see that students paste relatively often when constructing their programs during the first weeks. Our preliminary results indicate that on average one tenth of students' codebase is generated by pastes. On the positive side, most of these pastes are likely to be copied from each student's previous work.

As for compilation errors, the observed frequency distribution of different error types did not quite match earlier studies. There are several potential causes for this. Compared with the other similar work, the NetBeans environment compiles code continuously and also provides constant and immediate feedback when such a compilation fails. Learners are therefore more likely to instantly fix an error that was only just introduced in the code instead of noticing this later when they invoke a compile themselves. Second, any code generation features, such as, code completion which was used commonly, end up reducing the number of some types of errors. For example, filling in method calls via the completion suggestion menu will automatically add matching parentheses and a semicolon. Similarly, typing a lone opening parenthesis will automatically add a closing parenthesis after it. Furthermore, another feature that may have an impact is how NetBeans highlight the occurences of a variable when the caret is on one. This may allow learners to spot some typos but these often probably already come apparent during a compilation.

When analysing the compilation success percentages over the exercises (Fig. 3), we identified peaks. These peaks can e.g. indicate the difficulty of certain assignments as well as the structure of the handout. For example, the large peak at right hand side of the Figure 3 is caused by an exercise where students are introduced to writing their own methods; students receive a template of a method where they need to implement the printing of a passed variable – the students are already familiar with printing – and then call that method. Information on the difficulty of an exercise can be useful for example for constructing adaptive feedback [12].

Overall, based on the error reports in the works of Jadud and Denny et al, as well as based on our results, we can hypothesize that the compilation errors are a product of the used exercises, programming environment and the event logging granularity and thus, further studies in different contexts are needed.

5.1 Limitations of Study

As is natural for studies like this, the inherent limitation of our study is that the data comes from a single course. We have sought to remedy this a bit by including students from an open course instead of choosing students from a typically homogeneous computer science classroom. While this creates another selection bias, we believe that the population is less homogeneous than it would be if the population would have been drawn from the students at the University of Helsinki. Naturally, it would have been beneficial to be able to perform the study on multiple contexts, but no open data of students' programming process with the same granularity exists.

Another possible limitation is that it is possible that some of the students did not answer the survey detailing their background truthfully. However, as the survey was voluntary, and there were no rewards or other incentives that would have directed the students to participate in the survey, we can assume that most of the students were truthful. And, unfortunately, there is no easy way to determine whether the students were truthful or not.

As we analyzed the students' behavior within the NetBeans IDE, a possible limitation is that the student population was pre-selected. While we addressed this partially before, it is possible that the need to install a programming environment may create a bias in the population. While we agree that this is a limitation, we also provided single-click install packages that were tested in all major operating systems.

When considering the analysis of the approaches how students write their first lines of code, there are two limitations. First, in our case, there were two coders, and thus it is possible that intercoder reliability issues exist. However, the encoders sat next to each other during the encoding process, and voiced their thoughts and observations out loud throughout the encoding session. Another limitation is possibly the scope that we used for the analysis. As we studied only very simple cases, it is possible that students' struggles were missed. However, before the start of the encoding process, the encoders performed a rough analysis of the students' progress, and the exercises to be analyzed were deliberately chosen to (1) avoid taking too much space, (2) provide a view on students' growth, and (3) to be an example, where students first take multiple paths, but converge to specific approaches.

6. CONCLUSIONS AND FUTURE WORK

In this work, we analyzed how programmers with no previous programming experience write their first programs within an industry-standard programming environment. The analysis was done by qualitatively inspecting how students approach writing simple statements, and quantitatively inspecting the source code compilation statistics and the most typical errors that they face. Our results show no indication that students could not learn to program directly within a standard programming environment. On the contrary, students made active use of the features provided by the features such as programming environment -specific shortcuts and automatic completion of source code. An analysis of source code errors suggests that students' struggles are not as evident as they would be in environments such as BlueJ, where students need to run their applications to get feedback that indicates whether their code is in a compiling state or not. This likely means that methods such as the Jadud's EQ would not be as efficient on snapshot data, as most of the source code snapshots compile, and thus, the error pairings required by Jadud would be lost.

When considering our results on the compilation errors, and comparing them to existing work, it can be hypothesized that compilation errors are more of a feature of the used exercises, environments and data logging granularity. That is, the learning objectives of exercises transform students' actions, and thus, modify also some of the errors that are seen in the programming process. If the used environment provides feedback only on e.g. submission or save, it is likely that students start to use those features more, and thus, data points that otherwise would not be visible will get gathered. Naturally, also the logging granularity affects this; when logging every key-press, the data will naturally include more errors that are related to incomplete commands. Thus, more context-independent data-analysis studies are needed.

From the practical point of view, our results can be read in a way that starting with an off-the shelf IDE does not seem detrimental to students. Moreover, if a institution first uses an educational programming environment, and later moves the students away from such an environment, one may want to try starting with the off-the shelf IDE directly. We do note that students benefit from material and support, as they do in any course context. In our future work, we will (1) analyze the hypothesis that compilation errors are a product of the exercises, environment, and data granularity, (2) replicate this study on a larger dataset, and (3) perform a study where the behaviors of novice programmers are compared to those students that have explicitly stated that they have previous programming experience. Furthermore, (4) we will analyze how the copy-paste behavior is visible in these populations and will perform a more thorough analysis of the content that students paste.

7. REFERENCES

[1] J. Biggs and C. Tang. *Teaching for Quality Learning at University*. McGraw-Hill, 3rd edition, 2007.

[2] P. Blikstein. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pages 110–116, New York, NY, USA, 2011. ACM.

[3] N. C. Brown and A. Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[4] A. Christopher Strenta, R. Elliott, R. Adair, M. Matier, and J. Scott. Choosing and leaving science in highly selective institutions. *Research in Higher Education*, 35(5):513–547, 1994.

[5] S. Cooper, W. Dann, and R. Pausch. Alice: A 3-d tool for introductory programming concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, Apr. 2000.

[6] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pages 75–80, New York, NY, USA, 2012. ACM.

[7] G. Dyke. Which aspects of novice programmers' usage of an ide predict learning outcomes. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 505–510, New York, NY, USA, 2011. ACM.

[8] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 3–14, New York, NY, USA, 2009. ACM.

[9] K. Heinonen, K. Hirvikoski, M. Luukkainen, and A. Vihavainen. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 229–234, New York, NY, USA, 2014. ACM.

[10] J. Helminen, P. Ihantola, and V. Karavirta. Recording and analyzing in-browser programming sessions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*, Koli Calling '13, pages 13–22, New York, NY, USA, 2013. ACM.

[11] R. Hosseini, A. Vihavainen, and P. Brusilovsky. Exploring problem solving paths in a Java programming course. In *Proceedings of the 25th Workshop of the Psychology of Programming Interest Group*, 2014.

[12] P. Ihantola, J. Sorva, and A. Vihavainen. Automatically detectable indicators of programming assignment difficulty. In *Proceedings of the 15th Annual Conference on Information Technology Education*, SIGITE '14, 2014.

[13] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[14] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[15] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, ICER '06, pages 73–84, 2006.

[16] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[17] M. C. Linn and M. J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

[18] R. Lister. Concrete and other neo-piagetian forms of reasoning in the novice programmer. In J. Hamer and M. de Raadt, editors, *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, volume 114 of *CRPIT*, pages 9–18, Perth, Australia, 2011. Australian Computer Society.

[19] B. N. Miller and D. L. Ranum. Beyond pdf and epub: toward an interactive textbook. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 150–155. ACM, 2012.

[20] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science*

Education, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.

[21] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. ACM.

[22] D. Pritchard and T. Vasiga. Cs circles: an in-browser python course for beginners. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 591–596. ACM, 2013.

[23] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[24] M. M. T. Rodrigo and R. S. Baker. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 75–80, New York, NY, USA, 2009. ACM.

[25] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *TOCE*, 13(4):15:1–15:64, Nov. 2013.

[26] M. Vee, B. Meyer, and K. L. Mannock. Understanding novice errors and error paths in object-oriented programming through log analysis. In *Proceedings of workshop on educational data mining at the 8th international conference on intelligent tutoring systems (ITS 2006)*, pages 13–20, 2006.

[27] A. Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *Proceedings of the 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 498–499, 2013.

[28] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[29] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122, 2013.

[30] C. Watson, F. Li, and J. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 13th International Conference on Advanced Learning Technologies*, ICALT '13, pages 319–323, 2013.

[31] C. Watson and F. W. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*, ITiCSE '14, pages 39–44, New York, NY, USA, 2014. ACM.

[32] M. Yudelson, R. Hosseini, A. Vihavainen, and P. Brusilovsky. Investigating automated student modeling in a Java MOOC. In *Proceedings of The Seventh International Conference on Educational Data Mining 2014*, 2014.

Publication III.2

Petri Ihantola, Juha Sorva, and Arto Vihavainen

Automatically Detectable Indicators of Programming Assignment Difficulty

In *Proceedings of the 15th Conference on Information Technology Education (SIGITE '14).*

Automatically Detectable Indicators of Programming Assignment Difficulty

Petri Ihantola
Aalto University
Department of Computer Science and Engineering
Helsinki, Finland
petri.ihantola@aalto.fi

Juha Sorva
Aalto University
Department of Computer Science and Engineering
Helsinki, Finland
juha.sorva@aalto.fi

Arto Vihavainen
University of Helsinki
Department of Computer Science
Helsinki, Finland
avihavai@cs.helsinki.fi

ABSTRACT

The difficulty of learning tasks is a major factor in learning, as is the feedback given to students. Even automatic feedback should ideally be influenced by student-dependent factors such as task difficulty. We report on a preliminary exploration of such indicators of programming assignment difficulty that can be automatically detected for each student from source code snapshots of the student's evolving code. Using a combination of different metrics emerged as a promising approach. In the future, our results may help provide students with personalized automatic feedback.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

Keywords

automated assessment; programming assignments; assignment difficulty; personalized feedback

1. INTRODUCTION

In typical CS and IT curricula, introductory programming courses are among the first that students take. The rest of the curricula build on the skills learned in those courses, and indeed success in introductory courses affects whether students' continue with their studies or not [7]. It is not surprising that ways to improve introductory-level programming education has been under study for decades [21, 29].

Some decades ago, programming was a skill needed by a select few. It was often learned and taught in an *ad hoc* fashion, as educators often sought to replicate the ways in which they had themselves happened to learn to program, and there was no pressure to educate large numbers of graduates. From the 1980s onwards, programming has become increasingly mainstream, to the point that some countries have included programming education in primary school (e.g. [9]). At the same time, the practices and tools used to teach programming have evolved; these include novice-friendly programming environments [16] and microworlds [10], languages for beginners [20, 24], and program visualization tools [26], among others. Nevertheless, this evolution is arguably lagging behind the demand for more programmers and better programming pedagogy.

Pedagogical approaches to programming have been proposed in which students in read code and study many worked examples [17, 18]; software tools have also been designed to support activities such as code-reading and multiple-choice questions that help develop knowledge of important concepts. Such activities can be very useful in a complementary role, but if the goal is to learn to write programs, the pedagogy should be aligned with that goal [4] and must eventually include activities in which the students practice writing programs. In a university course or similar formal learning context, this means that the pedagogy needs to include *programming assignments.*

As with any form of practice, two key aspects of a programming assignments are feedback and student motivation. These aspects are connected, as good feedback can not only help the learner with the topic of the assignment but also increase the learner's motivation. Poor feedback, on the other hand, can make a learner less inclined to persist with a programming task or an entire course.

Feedback can be partially or fully automatic [28, 14]. Automatic assessment systems and intelligent tutoring systems bring benefits such as easy accessibility and low cost per student, which makes them particularly attractive in large classes with hundreds or thousands of students — and even more so in the context of modern massive online courses. The downside of many automatic solutions is that they fall short of a human tutor in terms of quality of feedback. A part of this problem is that the feedback provided by automatic systems is usually not personalized to take the learner and the learner's present knowledge into account. In order to provide better automatic feedback, we need to be able to judge how the individual learner (or group) relates to the assignment at hand. For instance, do they find it difficult? Trivial? Helpful? Ideally, a reasonably reliable estimate of such factors could be elicited automatically.

SIGITE'14, October 15–18, 2014, Atlanta, Georgia, USA.
Copyright 2014 ACM 978-1-4503-2686-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656450.2656476.

2. RESEARCH QUESTION

This article presents a study which is a preliminary exploration of factors that may influence the difficulty of programming assignments and metrics for automatically assessing those factors. More specifically, we explore the research question: *How do a learner's programming background and automatically analyzable programming behavior relate to the perceived difficulty of different programming assignments?*

The work is motivated by the current state of automated assessment systems for programming assignments: We believe that an appropriate next step in the automated assessment of programming assignments is the ability to provide feedback that is adjusted to fit particular students' needs and struggles. One aspect of this development is that feedback should be adjusted to match students' perceptions of assignment difficulty. Ideally, such feedback could be provided without constantly prompting students to assess the difficulty of the various assignments they work on.

The remainder of this article is organized as follows. First, we review the related literature in Section 3 below. Sections 4 and 5 outline our research methodology and present our empirical results and related discussion. Section 6 discusses some limitations of our work and possibilities for expanding on this exploratory study; Section 7 concludes the article.

3. RELATED WORK

Related work is explored through four themes. We start with theories of learning as we discuss the relationship between task difficulty, practice and motivation, which leads to the second theme of feedback. This in turn brings us to the third theme: software for automatic assessment. Finally, we consider those empirical studies within computing education research which resemble ours in that they have measured students' difficulties with programming assignments.

3.1 Task difficulty and practice

Expertise is not innate. It commonly grows through *deliberate practice*, that is, effortful activity whose purpose is to optimize improvement [11]. The importance of deliberate practice is reflected in programming courses around the world, which are designed around assignments that afford students with the opportunity to practice their programming skills on increasingly complex tasks.

Practice can be more or less effective. Ideally, the difficulty of an assignment matches the learner's existing knowledge and skills so that the learner is challenged to make use of their full cognitive capacity but is not overwhelmed by *cognitive load* [22]. Although the ideal is difficult to meet, not least because learners' prior knowledge varies, teachers may consider their students' expected learning trajectories and sequence assignments accordingly. Models of instructional design have been proposed to support these endeavors (e.g., the 4C/ID model [27]). The design and sequencing of assignments may be viewed as a form of *scaffolding* that aids the learner to make progress within their *zone of proximal development* [31] as they practice on tasks that they could not do without the help of the scaffolding.

Task difficulty impacts students' motivation in several ways. For instance, as per expectancy–value theories of motivation [2], assignments that are too easy are likely to have low perceived utility, while hard ones have a higher cost of completion, which reduces motivation unless they have been carefully designed to sustain interest. Excessive difficulty also contributes towards poor *self-efficacy* [3], which hampers further learning.

Another form of scaffolding that impacts motivation is the feedback that learners receive.

3.2 Feedback and motivation

Hattie and Timperley [12] argue that three main roles of feedback are to help a learner understand 1) the goals of learning, 2) the learner's own progress towards those goals, and 3) the activities that are needed to make better progress. For present purposes, the second role—the progress made by the learner—is the most salient.

A teacher or educational environment can help a student reflect on their progress by providing feedback that relates the student's performance to a particular goal or subgoal. Constructive feedback can improve self-efficacy. Constructive does not always imply positive, however, and feedback on progress should take into account the student's background and prior performance as well as the difficulty of the task. A beginner completing a difficult task should be applauded, but as Borich and Tombari argue on the basis of the literature, teachers who "show surprise at [students'] success, give excessive unsolicited help, or lavishly praise success on easy tasks are telling students that they lack ability" [5]. Such feedback can be detrimental to self-efficacy and motivation. Inappropriate feedback may also quickly cause students to learn to distrust the feedback-giving teacher or environment. The matter is, of course, complicated by the fact that an activity is not equally challenging to all learners.

3.3 Automatic feedback

An on-campus lab with an instructor and a small number of students is a setting that is well-suited to good, individualized feedback [8]. When such labs are not an option, or as a supplementary measure to them, feedback may be worked into course materials and programming assignments, which can be delivered online.

There is a robust field of research that seeks to improve the automatic assessment of students' solutions to programming problems [14]. Typically, automatic feedback is provided after students' take an action such as submitting a solution for assessment; the feedback often consists of information on the correctness of the solution and perhaps some additional information about observed deficiencies. The feedback may also praise the student for getting a good score or exhort them to make an improved attempt.

Two weaknesses of the typical approach discussed above are: 1) The feedback is "passive", as it is only presented when the student requests it, e.g., by submitting a solution, instead of being proactively offered, say, when the student is experiencing difficulty. 2) Feedback messages are based on the features of the submitted solution only, and are not influenced by other relevant factors such as the student's background or the difficulty of the task for the particular student. For instance, an experienced student may receive excessive accolades for a trivial assignment, which then undermines any praise received for more challenging ones.

3.4 Programming assignment difficulty

In this subsection, we briefly review some work similar to ours, that is, projects whose purpose has been to evaluate the difficulty of programming tasks.

Alvarez and Scott studied the relationship between the student-estimated difficulty of programming assignments and a number of metrics [1]. They used a survey that asked students to estimate difficulty twice, first after initially familiarizing themselves with an assignment and again after finishing it. The highest correlations to estimated difficulty were found using code metrics such as lines of code and the amount of control flow statements within the code.

Several threads of research exist that have utilized data recorded from the students' programming process. Although these studies generally have not focused on assignment difficulty, some of them have explored related phenomena. For instance, Jadud [15] proposed a formula for quantifying compilation errors, which has been used to identify students' course and assignment outcomes. In another study, Rodrigo and Baker [25] sought to identify students that are frustrated using both log data as well as observations from external observers. It is plausible that compilation errors and frustration do correlate positively with difficulty.

Another approach could be to estimate the cognitive load of students: cognitive load depends on both the intrinsic difficulty of a learning task and the prior knowledge that students bring to it. One way to estimate cognitive load is to use a suitable questionnaire. This approach, which is being explored in a programming context by Morrison et al. [19], has the benefit of using validated instruments and a solid theoretical basis, but since it requires a survey with multiple items, it is not suitable for our purposes. Another method also based on cognitive load is featured in a recent pilot study [6], in which the concept of "thrashing" was operationalized by measuring mouse clicks in an IDE; thrashing was taken to be an indication of (excessive) cognitive load. The results of the study demonstrated that different programming languages lead to different patterns of thrashing, which may be indicative of differences in difficulty.

In the present study, we seek to explore new metrics for automatically identifying which programming assignments different students find difficult. Our intention is to take one step closer to providing better, individualized, motivating automatic feedback that takes into account not only the student's program but also task difficulty as experienced by the particular student.

4. DATA AND METHODS

The data used in this study comes from an open online programming course offered by the University of Helsinki during Spring 2014. It is a six-week Java course in which students are taught procedural programming for the first three weeks and object-oriented programming for the second three-week period. The course is taught using an assignment-intensive teaching style, where majority of the work is done within a programming environment. Details of the course have been previously published in [30].

After each assignment, students could provide numeric feedback on the difficulty of the assignment. The difficulty was given on a scale from 1 to 5, where one stands for "easy" and five for "hard". In addition, the programming environment used in the course stored key-level snapshot data, that is, each key-press by a student while working on a programming assignment was recorded. None of the questions were mandatory, and students could turn off the key-level snapshot data gathering at will. At the beginning of the course, the participants were asked to provide details on their programming experience.

The data set used in this study contains information on 417 students. This is after we included only students who had provided details on their programming background, provided feedback on assignment difficulty on at least three occasions, and kept the key-level snapshot recording enabled. Overall, the included students submitted 31255 solutions to assignments and provided details on the difficulty of an assignment 11161 times. That is, in about 36% of the submissions, the participant also provided feedback on the assignment difficulty.

The snapshot data was processed to include a time stamp as well as information on compilation state, i.e., whether the source code in each snapshot compiles. This data was aggregated to provide information on the process that each student took to solve an assignment. More specifically, for each assignment that a student works on, we aggregated details on (1) the time spent on the assignment, (2) the number of keystrokes made, (3) the percentage of keystrokes and time in a non-compiling state, (4) the number of lines of code, and (5) the number of control-flow elements in the program (e.g. *if, else, while, for, return*). By "time spent on the assignment" we mean the overall time spent modified with by truncating any pause of over five minutes between keystrokes to only five minutes. The number of keystrokes and the percentage of time/keystrokes in a non-compiling state are also potential indicators of struggling to make progress; if a student spends more time in a state where the code does not compile, or takes more steps than others while solving the problem, the assignment may seem more difficult overall. Line and control-flow element counts measure code complexity, and have previously been observed to be decent indicators of perceived difficulty [1].

We used quantitative analysis to identify factors that explain programming assignment difficulty. Correlations between the students' perceived difficulty and factors were computed using the R statistics package [23].

5. RESULTS AND DISCUSSION

This section describes our results in three parts. First, we discuss the effect of programming experience on perceived assignment difficulty. Then, we consider the relationships between perceived difficulty and individual factors: time, number of keystrokes, compilation state, lines of code, and the number of control-flow elements. Finally, we look at combining the various factors.

5.1 Programming Experience

In order to evaluate the effect of prior programming experience, the students were split in two groups on the basis of on their background. Of the participants, 230 reported no previous programming experience, while 187 described at least some experience with programming. Figure 1 displays the average perceived difficulty of each assignment for the groups, as well as a combined metric for all participants. As a Shapiro-Francia test revealed that the populations do not follow a normal distribution, a Wilcoxon signed-rank test was used as the paired difference test to measure whether the population means differ.

For the population with at least some existing programming experience, the median difficulty of the assignments is 1.735, while for the population with no previous program-

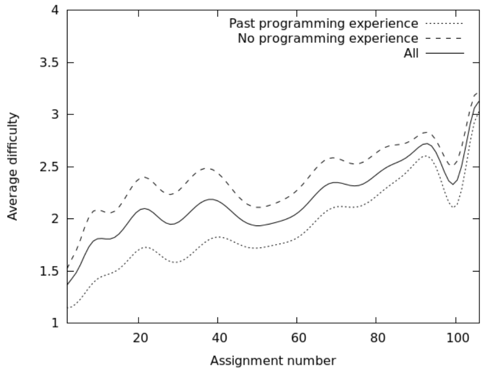


Figure 1: The means of students' estimates of the difficulty of the assignments. The curves have been smoothed for ease of viewing.

ming experience, the median difficulty of the assignments is 2.375. The populations are statistically different ($p < 0.01$), and thus there is, as one would expect, a difference in how the two populations perceive the difficulty of the assignments. However, as can be observed from Figure 1, the between-group difference in mean perceived difficulty diminishes towards the end of the six-week course. This trend suggests that the course taught skills that the more experienced students already had to some extent, and that the beginners partially caught up with the more experienced students.

5.2 Individual factors

Initially, an analysis was carried out to determine the correlations between difficulty and various other factors, each of which was considered separately. These factors were: time, number of keystrokes, proportions of keystrokes and time in a non-compiling states, lines of code, and count of control-flow elements. Table 1 displays these correlations for all participants as well as beginners and experienced students separately. For each factor, the table shows four values: one for all assignments, one for mathematically oriented assignments, one for open-ended assignments, and one for assignments with visual elements such as a given GUI that helps evaluate one's progress.

While a majority of the observed correlations are statistically significant, the correlation values are mostly medium-sized ($0.3 < r < 0.5$). In only two of the cases, the individual factors show a high correlation with difficulty ($r > 0.5$); both factors being time. The pattern of correlations appears to be largely similar for students with and without prior programming experience.

The correlations that we found between perceived difficulty and the number of lines of code as well as the number of control-flow elements were lower than the corresponding results reported earlier by Alvarez and Scott [1]. In their study, lines of code and control-flow elements had the highest correlations with perceived difficulty, whereas in our data, time on task and the number of keystrokes had somewhat higher correlations.

As Table 1 further shows, in most cases we found only low, largely insignificant correlations between perceived difficulty and the factors related to compilation status. The assignments with visual programming support constitute an exception to this trend, as a low-to-medium positive correlation was observed in these assignments.

Table 1: Correlation coefficients between perceived assignment difficulty and various metrics, grouped by assignment type (math-oriented vs. open-ended vs. visually supported). Correlations marked with an asterisk are statistically significant ($p < 0.01$).

Factor	All	Math	Open	Vis
All participants				
Time	.49*	.48*	.30*	.48*
Number of keystrokes	.44*	.42*	.27*	.39*
% states not compiling	.16*	.10	.03	.33*
% time not compiling	.03*	.09	.11	.20*
Lines of code	.36*	.27*	.24*	.26*
Control-flow elements	.35*	.26*	.22*	.38*
Programming experience				
Time	.54*	.45*	.36*	.45*
Number of keystrokes	.50*	.47*	.35*	.37*
% states not compiling	.15*	.10	.04	.27*
% time not compiling	.01	.11	.02	.13
Lines of code	.44*	.34*	.33*	.34*
Control-flow elements	.43*	.30*	.19	.36*
No programming experience				
Time	.46*	.48*	.25*	.53*
Number of keystrokes	.40*	.38*	.22*	.45*
% states not compiling	.16*	.10	.03	.36*
% time not compiling	.03	.08	.22	.20*
Lines of code	.33*	.25*	.18*	.23*
Control-flow elements	.31*	.25*	.25*	.45*

5.3 Combined factors

In the previous section, we considered individual indicators of assignment difficulty one at a time. To get an initial understanding of how these factors interact in the data set at our disposal, we applied a recursive partitioning to construct a decision tree of assignment difficulty. The model is based on the metrics presented in the previous section.

The model was built using the ctree implementation of R[1]. This method guarantees that the size of the tree is appropriate so that no pruning or cross-validation is necessary. A general description of the method is provided by Hothorn et al. [13].

The resulting decision tree is depicted in Figure 2. At each end node (leaf), a range of difficulty values is shown. This is the range of all the assessments of difficulty by students whose development snapshots matched the decision nodes leading to the end node. As can be seen from the figure, the tree is dominated by the amount of time that the student spent on the assignment. Although program size, complexity, and the degree to which the student maintained their program in a compilable state had an effect, students generally reported time-consuming exercises to be difficult.

6. LIMITATIONS AND FUTURE WORK

Our data comes from a particular programming course taught in a particular way in a Nordic country with a rather homogeneous population and high quality of education. Our results may be context-dependent. Indeed, as a part of the

[1] `http://www.inside-r.org/packages/cran/party/docs/ctree`

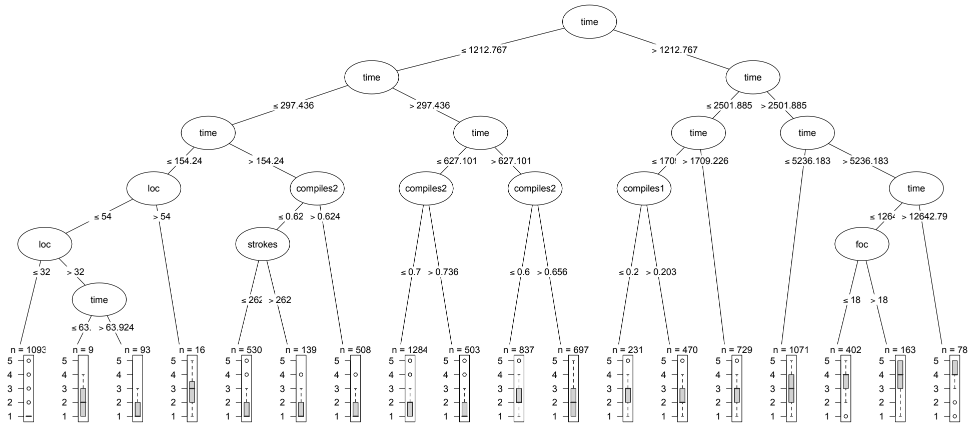


Figure 2: A decision tree of assignment difficulty constructed using recursive partitioning. *Loc* stands for lines of code and *foc* for flow-of-control count. *Compiles1* and *compiles2* stand for the proportion of compiling states and the proportion of time during which a program compiles, respectively.

present work, we re-examined the factors such as lines of code and number of control-flow elements that had the highest correlations with difficulty as reported by Alvarez and Scott [1], and while our work confirms that these metrics correlate positively with perceived difficulty, we found lower correlations for these factors than the other study. This difference in results suggests that the phenomena we have studied are at least in part, and perhaps quite significantly, dependent on context. Although a decision tree such as the one in Figure 2 may be useful in tailoring feedback in a particular context, other contexts need to be addressed separately. Future work may explore different programming courses, introductory or otherwise, and determine the extent to which our findings are transferable.

Our study draws on students' subjective assessments of difficulty. However, different students may have different interpretations of what "difficult" means. As pointed out in Section 5.3, much of "assignment difficulty" can be explained by the time it takes from students to do the assignment; similarly, Alvarez and Scott [1] reported that the size of the program was an important factor. A challenge in research such as ours, and one that we have not addressed in the present work, is teasing apart difficulty and workload, or at least determining the extent to which the distinction between the two is important for providing good feedback. The decision tree approach used above would easily accommodate additional variables, if necessary.

Another limitation is that although each assignment received a difficulty rating from at least 60 different students, providing the ratings was voluntary and we cannot rule out the possibility of an inherent self-selection bias. It is possible that the students that provided ratings are different from other students.

In this work, we split students in two groups on the basis of their prior programming experience. Future work could replace this simple model with a more fine-grained one so as to explore the variation among the experienced students.

To enable a critical examination of our results and facilitate follow-up studies, we have published our data set at `http://bit.ly/1oZnEKG`.

7. CONCLUSIONS

In this article, we have explored factors that relate to the perceived difficulty of programming assignments. The factors, excluding past programming experience, can be automatically detected from a stream of programming events—or keystrokes—that are performed within a programming environment. Our analysis suggests that the time spent on an assignment and the amount of programming events both have a medium to high correlation with perceived difficulty. Barely any correlation was found between perceived difficulty and the number of states that fail to compile or the length of time that the student's program is in a non-compiling state.

Although automatic feedback remains a far cry from what a good human tutor can provide, many students do not have convenient access to good human tutors, and any advance in automatic feedback is welcome. Our results show that metrics related to perceived difficulty can be automatically extracted from data that describes students' programming process. Automatic feedback systems can be adjusted to take task difficulty into account, which may improve the quality of feedback.

We conclude this article with some observations about the use of key-level data of student behavior. As a basis for assessing difficulty, this data has the benefit that it can be collected *in situ* and makes it possible to provide early, proactive feedback that the student does not need to explicitly request by submitting an assignment. An additional benefit, whose implications for automatic feedback may be explored in future work, is that such data provides details about students' programming process. Since, as Hattie and Timperley put it, "feedback is effective when it consists of information about progress, and/or about how to proceed" [12], key-level data has the potential to further enhance feedback.

8. REFERENCES

[1] A. Alvarez and T. A. Scott. Using student surveys in determining the difficulty of programming assignments. *J. Comput. Sci. Coll.*, 26(2):157–163, Dec. 2010.

[2] E. M. Anderman and H. Dawson. Learning with motivation. Routledge, 2011.

[3] A. Bandura. Self-efficacy: Toward a unifying theory of behavioral change. *Psych. Review*, 84(2):191, 1977.

[4] J. Biggs and C. Tang. *Teaching for Quality Learning at University*. McGraw-Hill, 3rd edition, 2007.

[5] G. D. Borich and M. L. Tombari. *Educational Psychology: A Contemporary Approach*. Longman Publishing/Addison Wesley, 2nd edition, 1997.

[6] S. Buist. Extending an IDE to support input device logging of programmers during the activity of user-interface programming: Analysing cognitive load. Bachelor of Science dissertation, The University of Bath, 2014.

[7] A. Christopher Strenta, R. Elliott, R. Adair, M. Matier, and J. Scott. Choosing and leaving science in highly selective institutions. *Research in Higher Education*, 35(5):513–547, 1994.

[8] M. Clancy, N. Titterton, C. Ryan, J. Slotta, and M. Linn. New roles for students, instructors, and computers in a lab-based introductory programming course. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 132–136, New York, NY, USA, 2003. ACM.

[9] Computing At School. Computing at school web site. `http://www.computingatschool.org.uk/`, n.d.

[10] S. Cooper, W. Dann, and R. Pausch. Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, Apr. 2000.

[11] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psych. Review*, 100(3):363, 1993.

[12] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[13] T. Hothorn, K. Hornik, and A. Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, 2006.

[14] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93. ACM, 2010.

[15] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84. ACM, 2006.

[16] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[17] M. C. Linn and M. J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

[18] R. Lister. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In J. Hamer and M. de Raadt, editors, *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, volume 114 of *CRPIT*, pages 9–18, Perth, Australia, 2011. Australian Computer Society.

[19] B. B. Morrison, B. Dorn, and M. Guzdial. Measuring cognitive load in introductory CS: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 131–138, New York, NY, USA, 2014. ACM.

[20] S. Papert. *Teaching Children Thinking (LOGO Memo)*. Massachusetts Institute of Technology, A.I. Laboratory, 1971.

[21] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ITiCSE Working Group Reports*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.

[22] J. L. Plass, R. Moreno, and R. Brünken, editors. *Cognitive Load Theory*. Cambridge Univ. Press, 2010.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[24] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[25] M. M. T. Rodrigo and R. S. Baker. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pages 75–80, New York, NY, USA, 2009. ACM.

[26] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):15:1–15:64, Nov. 2013.

[27] J. J. G. van Merriënboer and P. A. Kirschner. *Ten Steps to Complex Learning: A Systematic Approach to Four-Component Instructional Design*. Lawrence Erlbaum, 2007.

[28] K. VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011.

[29] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[30] A. Vihavainen, M. Luukkainen, and J. Kurhila. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 171–176, New York, NY, USA, 2012. ACM.

[31] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, 1978.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FI-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

A-2013-10 D. Entner: Causal Structure Learning and Effect Identification in Linear Non-Gaussian Models and Beyond. 79+113 pp. (Ph.D. Thesis)

A-2013-11 E. Galbrun: Methods for Redescription Mining. 72+77 pp. (Ph.D. Thesis)

A-2013-12 M. Pervilä: Data Center Energy Retrofits. 52+46 pp. (Ph.D. Thesis)

A-2013-13 P. Pohjalainen: Self-Organizing Software Architectures. 114+71 pp. (Ph.D. Thesis)

A-2014-1 J. Korhonen: Graph and Hypergraph Decompositions for Exact Algorithms. 62+66 pp. (Ph.D. Thesis)

A-2014-2 J. Paalasmaa: Monitoring Sleep with Force Sensor Measurement. 59+47 pp. (Ph.D. Thesis)

A-2014-3 L. Langohr: Methods for Finding Interesting Nodes in Weighted Graphs. 70+54 pp. (Ph.D. Thesis)

A-2014-4 S. Bhattacharya: Continuous Context Inference on Mobile Platforms. 94+67 pp. (Ph.D. Thesis)

A-2014-5 E. Lagerspetz: Collaborative Mobile Energy Awareness. 60+46 pp. (Ph.D. Thesis)

A-2015-1 L. Wang: Content, Topology and Cooperation in In-network Caching. 190 pp. (Ph.D. Thesis)

A-2015-2 T. Niinimäki: Approximation Strategies for Structure Learning in Bayesian Networks. 64+93 pp. (Ph.D. Thesis)

A-2015-3 D. Kempa: Efficient Construction of Fundamental Data Structures in Large-Scale Text Indexing. 68+88 pp. (Ph.D. Thesis)

A-2015-4 K. Zhao: Understanding Urban Human Mobility for Network Applications. 62+46 pp. (Ph.D. Thesis)

A-2015-5 A. Laaksonen: Algorithms for Melody Search and Transcription. 36+54 pp. (Ph.D. Thesis)

A-2015-6 Y. Ding: Collaborative Traffic Offloading for Mobile Systems. 223 pp. (Ph.D. Thesis)

A-2015-7 F. Fagerholm: Software Developer Experience: Case Studies in Lean-Agile and Open Source Environments. 118+68 pp. (Ph.D. Thesis)

A-2016-1 T. Ahonen: Cover Song Identification using Compression-based Distance Measures. 122+25 pp. (Ph.D. Thesis)

A-2016-2 O. Gross: World Associations as a Language Model for Generative and Creative Tasks. 60+10+54 pp. (Ph.D. Thesis)

A-2016-3 J. Määttä: Model Selection Methods for Linear Regression and Phylogenetic Reconstruction. 44+73 pp. (Ph.D. Thesis)

A-2016-4 J. Toivanen: Methods and Models in Linguistic and Musical Computational Creativity. 56+8+79 pp. (Ph.D. Thesis)

A-2016-5 K. Athukorala: Information Search as Adaptive Interaction. 122 pp. (Ph.D. Thesis)

A-2016-6 J.-K. Kangas: Combinatorial Algorithms with Applications in Learning Graphical Models. 66+90 pp. (Ph.D. Thesis)

A-2017-1 Y. Zou: On Model Selection for Bayesian Networks and Sparse Logistic Regression. 58+61 pp. (Ph.D. Thesis)

A-2017-2 Y.-T. Hsieh: Exploring Hand-Based Haptic Interfaces for Mobile Interaction Design. 79+120 pp. (Ph.D. Thesis)

A-2017-3 D. Valenzuela: Algorithms and Data Structures for Sequence Analysis in the Pan-Genomic Era. 74+78 pp. (Ph.D. Thesis)