View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Multi-faceted Support for MOOC in Programming

Arto Vihavainen, Matti Luukkainen and 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

Many massive open online courses (MOOC) have been tremendously popular, causing a stir in academic institutions. The most successful courses have reached tens of thousands of participants. In our MOOC on introductory programming, we aimed to improve distinctive challenges that concern most of the open online courses: allowing and requiring the participants to be more active in their online learning ("flipped-classroom"), demanding them to go deeper than typical CS1 course, and added incentives for participant retention by treating the course as a formal entrance exam to CS/IT degree. Our Extreme Apprenticeship (XA) method for programming education appeared to be successful in an online environment as well.

Categories and Subject Descriptors

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

General Terms

Experimentation, Management

Keywords

mooc, programming, extreme apprenticeship, automatic assessment service, formal acknowledgement

1. INTRODUCTION

In IT education, use of technology to support learning has evolved naturally. Most of the university level programming courses are hybrid courses offering both online materials and local lectures. Purely online courses in science and engineering are more rare, even though it has been said that they have "tremendous potential", yet "limited success to date" [14].

Reasons for limited success might lie in the fact that programming is a complex skill that requires rigorous practice [13]. Therefore, it has been stated that it is hard work to create purely online courses in the domain [7]. However, it has become apparent that the times are changing: "tremendous potential" has recently started to become more and more evident as freely available online courses, so-called MOOCs (massive open online courses) have gained momentum.

MOOCs are 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" [10]. As the word *massive* implies, MOOCs need to be inherently scalable. Moreover, they may share "conventions of an ordinary course, such as a predefined 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" [10].

Some prominent MOOCs, such as various classes from Stanford University, edX, Coursera and Udacity, have attracted tens of thousands of participants. Even though the retention rates may be very low [15], huge popularity in starting students shows that the time is ripe for MOOCs [6]. Courses in computer science and engineering can help to attract students into the field. In addition, easy-to-start web-based programming environments such as Codecademy allow people to get a taste of power of programming with a minimal initial threshold.

This paper presents *yet another MOOC* in introductory programming, and describes how we purposefully employed sophisticated pedagogical, technical and structural support to benefit our MOOC.

2. NOT YET ANOTHER MOOC

We claim that our MOOC is worth the effort in three important and distinctive areas: (1) the pedagogical method called Extreme Apprenticeship (XA) behind it has been particularly suitable for programming education [16, 9]. (2) Scaffolding of students' tasks using a purpose-built assessment solution and material combined with XA is especially suitable for self-study. (3) By coupling formal educational structures to support engagement to the course, we have been able to make the course more lucrative, yet more demanding and rewarding than many typical CS1 courses reported in current literary.

Contrary to the definition of a MOOC above, we deliberately wanted to reverse "no predefined expectations for participation, and no formal accreditation" to "clear reasons for participation, and clear formal accreditation".

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGITE'12, October 11–13, 2012, Calgary, Alberta, Canada.
Copyright 2012 ACM 978-1-4503-1464-0/12/10 ...$10.00.

2.1 Pedagogical approach: XA

The Extreme Apprenticeship (XA) method is based on Cognitive Apprenticeship [4, 5] and emphasizes guiding the students' working process and actual activity over "being taught". Core values in XA are [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 meaningful activities, i.e. exercises.
- Continuous feedback between the learner and the advisor. The learner needs confirmation that tells her that she is progressing *and* to a desired direction. Therefore, the advisor must be aware of the successes and challenges of the learner throughout the course.

We have completely left out lectures and focused on providing high-quality learning material, exercises, and enough XA lab guidance in our recent CS1 courses. In XA labs the students are scaffolded by course advisors as they work on the exercises. Scaffolding refers to supporting students in a way that they are not given direct answers, rather, just pushed into a direction to discover the answers themselves.

An important factor is that XA is as "genuine" as possible. Therefore, learning to program starts with installing industry-strength programming tools; we use NetBeans as our IDE of choice as it is freely available and provides good debugging and testing capabilities. Using recognized tools emphasizes that the learners are on a path to become true professionals.

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 [16]. Every student practices with dozens of simple exercises already during the first week, and continues working on more demanding exercises throughout the course. Scaffolding is built into the exercises and material, which makes it easier for the student to proceed to a favorable direction. An important factor is that learning achievements are made visible to the student, thus boosting the motivation to continue.

2.2 Test My Code (TMC)

XA is by definition direct one-on-one interaction between the student and the advisor. Although XA can be scaled to hundreds of students with the use of efficient resource and personnel allocation [9], it is evident that XA can benefit from automated tools. In order to allow advisors to focus their time to actual scaffolding tasks, we have crafted a scalable automatic assessment solution that allows building scaffolding into programming exercises, provides full bookkeeping within several simultaneous courses, and has course management functionality.

Test My Code (TMC) is a bundle that contains several parts: a NetBeans programming environment plugin that integrates seamlessly to the normal programming workflow, a scalable assessment server that can be deployed to cloud environments, and a purpose-built domain-specific language (DSL) that is used by the course staff for building exercises with built-in scaffolding messages. More specific tests are crafted using unit tests, which means that there are practically no limits on what can be tested.

As new exercises are released, the TMC NetBeans plugin downloads them as programming projects into the IDE. Each project contains exercise-specific tests and, in some cases, code that is given as a starting point for the exercise.

Contrary to most of the automatic assessment systems [8], students can test their solutions locally using the programming environment and receive direct scaffolding inside the IDE. Tests are quick to run and can be executed as often as the student wishes. Students are also able to see the source code of the tests. However, if we do not want to expose the tests to students, TMC also supports creating *hidden tests* that are only available on the assessment server.

We use TMC in our on-campus CS1 courses. However, as TMC offers bookkeeping and scaffolding, it was only natural to open our course in a completely online form as a MOOC. When comparing our MOOC with our on-campus CS1 course, MOOC participants do not have live one-to-one advisors guiding them. Other than that, our MOOC is exactly the same as the on-campus course.

2.3 Extending incentives for participation

We wanted to give the option for formal acknowledgement for participants working through our CS1 course as our university students do. This was done in two parts; (1) we provided five partial exams for students in K-12 institutions that were graded at our university, and (2) we offered a full admission to CS/IT degree program for a top-tier university to every high-performing student of the course.

The track for applying to the computer science department required the ability and motivation to complete most of the exercises, to pass a monitored programming exam and an interview conducted by the faculty of our department, and to be eligible for university-level education (e.g. finishing or finished secondary education).

From our perspective, admitting the top-performers can be seen as a "win-win-win-situation". As they have participated in the course, exam and the interview, we know that (1) they are able to perform in a university level course, (2) they have tried programming and noticed that they want to study computer science, and (3) they have a "flying start" to their studies as they have already finished CS1 with very high marks.

3. COURSE ORGANIZATION

With the introduction of XA our CS1 "staff" has become relatively large. We have around 20 persons associated with the course, almost all of them students, so it has been only natural to have a set of well-defined roles, responsibilities, and practises.

3.1 Course Personnel

As XA is a form of apprenticeship education, the "pyramid" of the stakeholders is essential in organizing the course: there are *masters* (tenured teachers working also as advisors) that are on the top of the pyramid, crafting material and exercises, coordinating and controlling the operation; *journeymen* (paid advisors that contribute to exercises and help the students with explicit responsibilities); *apprentices* (unpaid advisors among fellow students with limited responsibilities); and finally, *students* of the course (potential apprentices of future courses)

Working as an apprentice is a part of our non-mandatory CS studies, that focuses on the importance of "soft skills" and coaching. Apprentices receive credit that is relative to the amount of advising done, typically between 1 and 3 cred-

its[1]. Using the apprenticeship system allows us to provide teaching and coaching experience for many of the students, as well as give them responsibility. Giving formal credits for the apprentice work is an important incentive, and it enables us to establish an understanding of what is required from the otherwise unpaid advisors.

The journeymen are selected for each course instance using an open call. Usually, they have previous teaching experience or have been working as apprentices in the past CS1 courses. In addition to the extra responsibility and income, it is important that the journeymen proceed in their own studies as they serve as role models for students and apprentices.

3.2 Working Process

Because of the rapid feedback cycle in XA labs, our course content and working process is elastic; whenever we notice something that can be improved, we act as soon as possible. Typically, we improve the material weekly, and the material is at least partially rewritten during each course.

The material and exercises are developed by masters with help from journeymen. As developing new exercises includes creating tests that scaffold students, it is important that new exercises are tested thoroughly before publishing them to the course population. To verify the thoroughness of the tests, we have created a "Alpha-Beta-Open" release cycle for verifying that the content is ready for MOOC release.

"Alpha-Beta-Open" means that prior to releasing the material to MOOC participants, the material is tested in required lengths by the journeymen and masters, and gradually released to the apprentices. Each member of the teaching staff, be it an apprentice, journeyman or a master, works as an *alpha-tester* before the release to our CS1 course. Alpha-testers work out the material and exercises and seek out possible issues. The issues are gathered in a helpdesk-like ticketing system that is processed by the journeymen and masters.

Once alpha-testing is finished and the found issues are resolved, the material is released to our CS1 course. The course is set up using XA so that the students work on the exercises in XA labs under guidance from advisors. This provides us another feedback point for improvement. Essentially the students in the university course are the *beta-testers*. Once enough students (over 10) in our CS1 have done the exercises without noticeable problems, the exercises and material are published to MOOC participants.

If problems are noticed after the exercises are published to the public, fixing them is still relatively easy. Updated material is easily republished, and TMC has built-in functionality that can automatically update the students' exercises to contain the latest fixes.

4. EXERCISE MATERIAL

As most of the introductory programming books and lecture materials are centered around language constructs and fail to present actual working process [12], we have created our own material. Our material is built around the exercises and emphasizes the actual working process using worked examples [3] and process recordings [2]. Both worked examples and process recordings emphasize *how* a program is crafted using stepwise subtask division: one must always start small to grow big.

[1]In our system one credit generally corresponds to 20-30 hours of work

Our semester-length Java CS1 course covers topics typical to most programming courses: assignment, expressions, terminal input and output, basic control structures, classes, objects, arrays and strings as well as object-oriented programming features such as interfaces, inheritance and polymorphism. File I/O, exceptions and GUIs are also covered, and essential features of Java API, such as lists, maps, and sets, receive tons of practise.

Best programming practises are emphasized as well; use of meaningful variable and method names, refactoring existing code into smaller methods, using the single responsibility principle, and using automated tests. Basic algorithmics, such as sorting and searching, are also covered.

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; individual effort plays the key role. In order to provide support for the students' working process, scaffolding the exercises is required.

4.1 Scaffolding in Exercises

Exercises form the core of our course: there are a total of 170 exercises that are split into a total of 373 tasks. Learning objectives are embedded into the exercises, and the material is built around the exercises to maximize scaffolding. The learning material is constructed so that new topics are immediately applicable to following exercises.

Most of the exercises are composed of small incremental tasks that together form bigger programs. Incremental tasks imitate a typical problem solving process. Students explicitly practise programming, but are constantly influenced by the written out thought process behind 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 that provide further support for verifying correctness of the program. This allows the student to confirm that she is proceeding to the correct direction.

In addition to the structural support from the material, students receive scaffolding from TMC. Exercise-specific tests are built in a way that gives direct support to the incremental nature of the exercises. Tests are structured so that the students can focus on progressing in small steps even within a single task.

Typical tasks may require implementing a class, writing method bodies that correspond to given signatures, and verifying that method outputs are correct for a set of inputs. Typically, the working process continues with an increment, e.g. by forming another class that perhaps uses the previously implemented class. In a way the students' workflow reminds the workflow in Test Driven Development [1] except that the tests are often readily given to student[2]. A clear metalevel motivation to the incremental style is to guide students to a similar working process that good professional programmers use: progress in small steps and after each step ensure that what you did works correctly.

Our experiments indicate that the incremental "scaffolded" process backed with tests works reasonably well. However, there is one challenge we still need to solve: with the introduction of TMC, some students have started to rely too

[2]We also have exercises where the students must create unit tests.

much on the tests, and do not write spontaneous main-programs of their own for testing. Creating small test programs for trials and debugging is extremely important since if a test fails, the corrective actions are not always trivial despite the fact that TMC tests provide rather good diagnostic messages.

After the students have been honing their skills with the scaffolded exercises, scaffolding is faded and students work on more open assignments. Open assignments let the students to design the internal program structure freely, but still provide support e.g. by defining the UI in a relatively strict manner. Depending on the learning objective of the exercise, it may be split into required functionalities providing additional scaffolding.

Open assignments are intentionally complex enough so that programming a solution to a single class will cause chaos, but simple enough so that using an "implement a single requirement, refactor if needed" -approach will end up with a nice object design. They usually describe a well-known domain (e.g. airport, airplanes), that helps students to grasp and design initial domain objects.

Although the majority of the exercises are scaffolded, each week contains open exercises as well. Having non-scaffolded open exercises in the course is of importance since our goal is that every student should be capable of performing elementary program design and problem solving independently.

5. SAMPLE EXERCISE

When students are learning to program, it is important that they are shown the process of creating a program step by step. The following Movie recommender -exercise is an example of a scaffolded exercise that supports the student in crafting a working movie recommender. The exercise has been inspired by the personalized book recommendation system presented as one of the nifty assignments in SIGCSE 2011 [11].

In addition to the following description, the description contained small program snippets that could be used for verifying the program functionality at specific stages, as well as reflective narratives. We have omitted them and other details such as packages due to article size constraints.

Movie Recommender

In October 2006, a corporation called Netflix promised 1 million dollars to the person or the group that would be able to create a program, that is 10% better at making personalized movie recommendations than their existing program. The competition was finished in Sept. 2009 (http://www.netflixprize.com) – unfortunately we did not win.

In this exercise, we build a program for recommending movies. The application can recommend movies based on overall and personal ratings. First we will create necessary domain objects, and then start crafting a database for storing ratings. Once we can store and retrieve ratings, we will build the actual recommender.

Task 1: Person and Movie Create classes Person and Movie. Both classes must have a public constructor that takes a name as a parameter, and a public method getName() that returns the name. In addition, create a toString()-method for both classes that returns the name that was received in the constructor, and override the existing equals- and hashCode-methods.

Task 2: Rating Create an enum-type class Rating that has the following values: terrible (-5), bad (-3), not seen (0), neutral (1), good (3), awesome (5).The class must have a public method called getValue() that returns a specific int-value.

Task 3: RatingDatabase (1) Create a RatingDatabase class. It should provide the following public methods: addRating(Movie movie, Rating rating) that adds a rating to the given movie, movieRatings() that returns movie specific ratings as Map<Movie, List<Rating>>, and getRatings(Movie movie) that returns the ratings for the given movie as a list.

Task 4: RatingDatabase (2) Add the following public methods to the class RatingDatabase. Method addRating(Person person, Movie movie, Rating rating) adds a new rating to the given movie done by the person, getPersonRatings(Person person) returns all ratings made by the given person as Map<Movie, Rating>, getRating(Person person, Movie movie) returns the rating for the movie by a given person. If no such rating exists, return rating *not seen*. Method getRaters() returns a list of persons that have added ratings. Each person may rate a specific movie only once; existing ratings are overwritten.

The functionality implemented in the previous task must not break.

Task 5: MovieComparator Create a MovieComparator class that implements interface Comparator<Movie>. The class should receive a Map<Movie, List<Rating>> as a constructor parameter, and should allow sorting movies based on their rating averages in descending order.

Task 6: Recommender (1) Create a MovieRecommender class that takes RatingDatabase as a constructor parameter. Add a public method recommendMovie(Person person) that recommends a movie to a person. At this point, recommend only the movie that has the highest average rating, and has not yet been seen by the given person. If no such movie exists, return null.

Task 7: PersonComparator Create a class PersonComparator that implements the interface Comparator<Person>. The comparator takes Map<Person, Integer> as a constructor parameter, and must allow sorting persons based on the map value in descending order.

Task 8: Recommender (2) When the system contains movie ratings from persons, we have knowledge on their movie taste. Extend the method recommendMovie(Person person) so that it gives a personalized recommendation if a person has rated movies. If the person has not given any ratings, the recommender should recommend a movie based on rating average.

Personalized ratings should be based on the similarity of ratings by a person when compared to other raters. Let us consider an example with 3 persons; Tom, Dick and Harry. Tom has rated movie A good (3), movie B terrible (-5). Dick has rated movie B terrible (-5), and movie C good (3). Harry has rated movie B good (3) and movie C good (3). The person-wise similarities are calculated based on the similarities of given movie ratings.

If Tom wants to have a movie recommended to him, we calculate the similarity of each person in relation to Tom. Similarity of Tom and Dick is 25 (-5 * -5), as they both have seen movie B and given it the rating terrible. The similarity of Tom and Harry is -15 (-5 * 3) as they also have both seen the movie B – Harry rated it good. As the similarity between Tom and Dick is higher, movies rated as good or excellent by Dick should be recommended to Tom.

Implement the described functionality.

The movie recommender exercise above is done towards the end of the course, as it contains object design, use of maps and lists, and requires basic algorithmic thinking. The exercise was one of five larger exercises in the week where it was released.

6. CRAFTING TESTS

Creating pedagogically sound tests for the exercises requires both expertise and experience. As none of the required classes or packages usually exist in the exercise base that is downloaded by TMC, scaffolding needs to be built

starting from nothing. We start scaffolding by helping with the very basic properties, e.g. does a required package exist, does a required class exist, is the class public and so on. Once a class is known to exist, the next step is to start verifying the existence of required methods with sensible visibility settings followed by working on the basic functionality of the methods, one by one.

TMC provides a DSL for creating scaffolding for basic properties very easily. In addition to the basic verification and scaffolding, we have created lots of tests for exercise-specific scaffolding; roughly over 1300 tests with over 4000 messages were crafted during the MOOC experiment.

Additional scaffolding is done via rigorous use of reflection. In the movie recommender exercise students are corraled into the right direction with questions like "what happens if the person has not rated a movie?" and "are you sure that the movie comparator produces results in correct order?". Each task has usually several tests that guide the students' working process as well as the student towards a more proper solution.

The testing of open exercises is usually done with a mixture of reflection and input-output testing. Reflection is used to verify consistent object design, for example in specific domain-related problems one may verify that the student has crafted classes that resemble assumed domain objects. Actual functionality testing is done using black-box input-output -testing where most of the possible execution paths as well as error cases are tested.

Testing is not always easy. For example one of our GUI-related exercises requires that the student draws a smiley that looks similar to an example image using only 5 rectangles. Actual testing requires mocking the Java's Graphics-object to verify the number of used rectangles, and approximating distance from the drawn image to an existing image using a %-match ratio needed for acceptance. We also had interactive GUI exercises; one of the more demanding ones scaffolded the students in building the classic snake game from scratch.

Crafting scaffolding into the exercises was considered rewarding by the course personnel. Even though some of them had experience from the industry, there was still lots to learn. It was essential that the staff was good at playing the "what could the student do next that would make the provided scaffolding break up?" -game. We feel that this was only possible due to the presence in local XA labs.

7. DATA FROM THE MOOC

We had a total of 417 registered participants in our MOOC using basic word-of-mouth advertising and contacting K-12 institutions. The low number of participants when compared to more established MOOC providers such as Udacity and Coursera can be partially explained by our relatively small language area; we offered the course in our native language with K-12 institutions specifically in mind.

When registering to the course, one did not need to fill in any personal details; we only required desired nickname and a contact email-address. Out of the 417 registered, 405 started working on the exercises and made at least a hello-world application. During our initial survey three weeks into the course, 67 students had indicated that they were applying for full admission – majority of the participants were on to "check out this new MOOC thing".

MOOC drop-out rates are typically very high, and hard

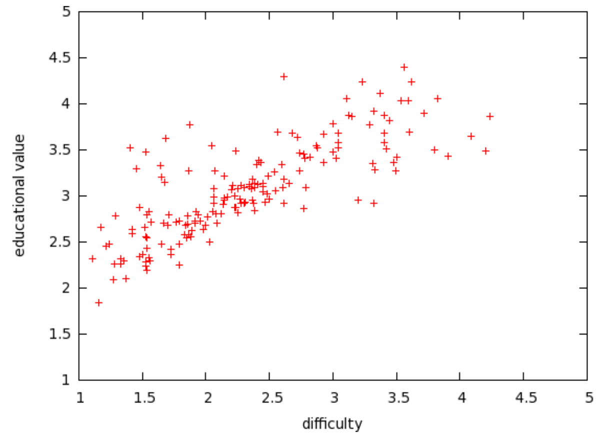

Figure 1: Participant feedback on exercise difficulty plotted against educational value.

figures that could be used for comparison are hard to come by. During our semester-length MOOC where 405 participants programmed at least one program, 329 participants did over programming 20 tasks, 256 continued to program over 50 tasks, and 187 over 100 tasks. After 6 weeks one third of the participants were still actively programming. A bit under 100 participants did over 80% of the tasks, and 70 participants finished over 90% of the tasks.

During the course we gathered voluntary feedback. The participants could provide numeric feedback on the difficulty (1 = easy, 5 = hard) and educational value (1 = low, 5 = high) for each exercise directly from TMC. The students also had an option for adding direct verbal feedback. Only 3 of the exercises had an average difficulty over 4; all of them were at least partially in the wrong place.

We observe a clear correlation between the difficulty and educational value of our exercises, see Figure 1. Our goal was to have each week start easy and early, to addict the students, and build the exercises so that the students could work in their zone of proximal development [16, 17].

In addition to the built-in feedback mechanism, we had setup a IRC channel that served as the main peer-support forum. Because of the clear timeline in the course, it proved to be a solid support channel throughout our MOOC. Our course page provided a web-interface for IRC access, and during the MOOC, we estimated roughly 250 individual participants. Several faculty members were also present – this is only natural given that they wanted to receive feedback and see how the course unfolds.

Most active hours were during the evenings. The IRC activity was relatively low on Saturday, even though Saturdays and Sundays are not typically schooldays/workdays in Finland. High activity on Sundays can be explained by the fact that the submission deadlines for participants were set for Sundays nights. We observed small peaks in the early Wednesdays of the early course. At a further inspection, this was due to our visible, live scoreboard for every nickname for the completed exercises: some participants started to compete on who finishes the exercise set the fastest. This effect was very prominent in the IRC channel in the early stages of the course, as the tasks were relatively straightforward for more experienced participants, but faded away as the course progressed.

8. CONCLUSIONS AND FUTURE WORK

We described how we offered our CS1 free for everybody as a MOOC using sophisticated pedagogical, technical and structural support. The initial feedback from the course participants has been fruitful, and we have been able to witness and gather several completely unsolicited, truly spontaneous testimonials among the flow of the IRC discussions (translated by the authors):

> *I must praise a bit before I continue. It's been a really good idea to organize this course! I've never programmed before, but have always been interested in programming. Starting to learn programming was made really easy*

> *I've learned more about programming on mooc than ever before*

> *After mooc one starts to understand why object-oriented programming was invented*

> *Right now I just want to quit my work and get back to studying*

> *This is the most in-depth programming course that I've ever had .. absolutely merciless*

The first comment was given early in the course and is in line with the "start early, start small" -view. The second and third comments were given mid-way through the course when the participants had been doing object-oriented programming for a few weeks. Scaffolded exercises where the problem had been divided into smaller tasks provided support, conveyed the working process, and described how problems should be solved using object-oriented programming. The second and fifth comment displays that people with existing programming background were also able to learn new things.

Fourth comment indicates the motivation boost that one receives from solving the exercises, and the last comment puts the demand-level of our approach into words: "absolutely merciless".

Our initial experiments have been successful: we have been able to create a model and tools for crafting a MOOC from our CS1 course, and have been able to provide the course for free for everyone. Our MOOC has not been a watered down version of our internal course, which may have increased the observed drop-out rates.

To our understanding, the drop-out rates have been similar to other offered MOOCs. In the fall of 2012, we have 41 starting "MOOC students", which provides an interesting chance of monitoring how they relate to the students admitted in a normal way. We are also investigating reasons on why participants have dropped out midway through the course, and are performing data-analysis on the solutions crafted by the MOOC participants.

As our national K-12 curriculum does not include IT related studies, many of the students miss the opportunity to learn programming. We provide support for basic programming education to any K-12 -school free of charge. The formally acknowledged MOOC approach with partial exams does not demand actual programming knowledge from teachers or the school faculty, and due to the nationwide technology programme any school has computers required for the course. Additionally, majority of the students have computers at home.

9. REFERENCES

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

[2] J. Bennedsen and M. E. Caspersen. Reflections on the teaching of programming. chapter Exposing the Programming Process, pages 6–16. Springer-Verlag, Berlin, Heidelberg, 2008.

[3] 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.

[4] 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.

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

[6] A. Fox and D. Patterson. Crossing the software education chasm. *Commn. ACM*, 55(5):44–49, May 2012.

[7] J. Gal-Ezer, T. Vilner, and E. Zur. The professor on your pc: a virtual cs1 course. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 191–195, New York, NY, USA, 2009. ACM.

[8] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proc. of the 10th Koli Calling Int. Conf. on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[9] 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.

[10] A. McAuley, B. Stewart, G. Siemens, and D. Cormier. The mooc model for digital practice. 2012.

[11] N. Parlante, J. Zelenski, K. Schwarz, D. Feinberg, M. Craig, S. Hansen, M. Scott, and D. J. Malan. Nifty assignments. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 491–492, New York, NY, USA, 2011. ACM.

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

[13] 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.

[14] J. Subhlok, O. Johnson, V. Subramaniam, R. Vilalta, and C. Yun. Tablet pc video based hybrid coursework in computer science: report from a pilot project. *SIGCSE Bull.*, 39(1):74–78, Mar. 2007.

[15] S. Thrun and D. Evans. Georgia tech talk quoted in computing education blog, 2012.

[16] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *SIGCSE '11: Proceedings of the 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.