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 %.

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

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.