Recalling Programming Competence

Jens Bennedsen
Engineering College of Aarhus
Dalgas Avenue 2
DK-8000 Aarhus C, Denmark
jbb@iha.dk

Michael E. Caspersen
Department of Computer Science
Aarhus University
DK-8000 Aarhus C, Denmark
mec@cs.au.dk

ABSTRACT

Programming is recognised as one of seven grand challenges in computing education and attracts much attention in computing education research. Most research in the area concerns teaching methods, educational technology, and student understanding/misconceptions. Typically, evaluation of learning outcome takes place during or immediately following the educational activity. In this research, we conduct a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place. Our population consists of ten students who have taken an introductory object-oriented programming course 3, 15, or 27 months prior to our study. None of the students have been exposed to programming in the intervening period. As expected, our research shows that syntactical issues in general hinder immediate programming productivity, but more interestingly it also indicate that a tiny retraining activity and simple guidelines is enough to recall programming competence and overcome syntactical issues.

Categories and Subject Descriptors

K3.2 [**Computers&Education**]: Computer and Information Science Education—*computer science education, information systems education*

General Terms

Experimentation, Human Factors

Keywords

CS1, object-oriented programming, remembering

1. INTRODUCTION

For many years there have been a massive interest in program education research and development. Teaching methods, materials, and educational technology have been developed to help students better learn how to program. Some of these innovations have been systematically evaluated for their impact, but in general the measurement of success is defined by how well the students perform at the final exam or at tests during the course. The quest for success indicators is an example of such studies (see e.g. [8, 42, 7]), and so are studies that evaluate educational technology (see e.g. [38, 24, 21]). Evaluating the impact immediately after the course, is of course both interesting and relevant, but in general the goals of our teaching is not only that the students perform well at the final exam, but that the students achieve relevant and lasting programming competences.

Computing competences are becoming relevant in many fields; consequently, many students who will not major in computer science will be required to take an introductory computing course [18]. Many introductory computing course has programming as a core activity and learning goal, and for good reasons since programmability is the defining characteristics of the (digital) computer. This is also echoed in the ACM/IEEE curriculum recommendations. Currently, a revision and enlargement of the curriculum recommendations is under way, broadening the scope from traditional computer science to the broader field of computing [35], from Information Systems [16] to Computer Engineering [37].In e.g. "the model curriculum and guidelines for graduate degree programs in information systems" [17] it is noted that *Students entering the MSIS program need the content of the following courses ... programming* (p.138).

We forget things. The cognitive structures that store facts and schemes typically become less accessible over time, and forgetting is more likely to take place when memory elements are not accessed and used [9]. By fitting data from several experiments in cognitive psychology, Woodworth [45] created the so-called forgetting curve, see figure 1. Accordingly, it should be expected that students do not have the same competences say one year after an exam as they had right after the exam.

In our current research, we are particularly interested in studying the durability of programming competences achieved in an introductory object-oriented programming course for non-CS majors. We have conducted a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place. Our population consists of ten students who have taken an introductory object-oriented programming course 3, 15, or 27 months previous to our test. None of the students, who are

Koli Calling '09 October 29 - November 1, 2009, Koli, Finland

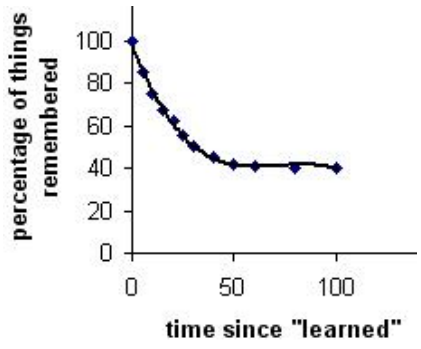


Figure 1: Classic shape of the forgetting curve (Woodworth, 1938).

majors in bio-technology, have been exposed to programming in the intervening period.

The remaining part of the article is organised as follows: Section two describes related work primarily in cognitive psychology. In section three and four we describe the instructional design of the introductory programming course. Section five presents our hypotheses and research questions, and section six presents our research design. In section seven we describe and analyse our observations. Potential future work is described in section eight, and section nine is the conclusion.

2. RELATED WORK

This section describes the related work. It focuses on two things, work in the area for forgetting and work in the area of remembering programming competences.

2.1 Memory

The memory is fallible. The fact that we gradually forgets was first documented by Ebbinghaus [14] in a study where he first tried to learn nonsense syllables and then tried to remember as much as possible at various delays after the learning. His conclusion was, that there was a rapid drop-off in retention in the beginning and then a more gradual drop-off later. As he wrote: *One hour after the end of the learning, the forgetting had already progressed so far that one half the amount of the original work had to be expended before the series could be reproduced again; after 8 hours the work to be made up amounted to two thirds of the first effort. Gradually, however, the process became slower so that even for rather long periods the additional loss could be ascertained only with difficulty. After 24 hours about one third was always remembered; after 6 days about one fourth, and after a whole month fully one fifth of the first work persisted in effect* (section 29, [15]). Ebbinghaus found that a complex logarithmic function described his data. Later it has been shown [43] that a power function $y = \alpha t^{\beta}$ better describes the relation between time and remembering. The values of α and β relies upon the actual person and the "thing" to remember.

Apart from an interest in forgetting, Ebbinghaus was also interested in the effect of repeated learning. He found that *The relation is quite similar to that described in Chapter VI [the relation between time and forgetting] as existing between the surety of the series and the number of its repetitions* (section 31, [15]).

Relearning affects forgetting. As Schacter [33] notice *it is known, for instance, that retrieving and rehearsing experiences play an important role in determining whether those experiences will be remembered or forgotten* (p. 184). The current memory model is actually more complex than a simple correlation between recall and remembering. Loftus [25] have found four major reasons why people forget: retrieval failure (memory traces decay over time), interference (memory may compete and interfere with other memory) , failure to store (e.g. details may be filtered out) and motivated forgetting (we want to forget e.g. traumatic things). As Anderson, Bjork and Bjork [2] notice *a striking implication of current memory theory is that the very act of remembering may cause forgetting. It is not that the remembered item itself becomes more susceptible to forgetting; in fact, recalling an item increases the likelihood that it will be recallable again at a later time. Rather, it is other items — items that are associated to the same cue or cues guiding retrieval — that may be put in greater jeopardy of being forgotten.* (p. 1063). According to Anderson, Bjork and Bjork the reason for this is three assumptions on how the memory work

the competition assumption Memories associated to a common cue compete for access to conscious recall when that cue is presented,

strength dependence assumption a cued recall of a memory will decrease as a function of increases in the strength of its competitors,

retrieval-based learning assumption Recall of a memory enhances subsequent recall of that memory.

The knowledge of forgetting have inspired many (primary) schools to evaluate their school calendar [13]. In general there seems to be an impact of a calendar model with many small breaks as opposed to one long summer break since students tend to perform better on tests with many small breaks rather than one large break. The effect of forgetting was notable particularly with respect to math facts and spelling. Findings in cognitive psychology suggest that without practise, facts and procedural skills are most susceptible to forgetting [12]. The categories of facts and procedural skills most likely encompass the idiosyncrasy of programming language syntax and programming skills which is the focus of our research.

2.2 Learning to Program

Many approaches to introductory programming education have been proposed including a procedures early approach [29], a top-down approach [19, 30], a graphics approach [26]. Even within introductory object-oriented programming, many different approaches exist: objects early [1], interfaces early [34], GUIs early [44], concurrency early [31], events early [39], components early [20], etc.

All of these articles about introductory programming education describe different (groups of) people's approaches. However, many are in the "Marco Polo" style of reporting research in introductory programming [41]; or, to be more precise, they argue that a certain approach is better than others based on the assumption that certain learning outcomes should be promoted.

To properly evaluate the long-time learning effect of a program course, we must take as starting point the intended learning outcomes (ILO) of the course. Whether the ILO focus on special features of the programming language, the process of program development, or something else, has an impact on how we must test and recall programming competences. In the following section we will describe the ILO for the introductory object-oriented programming course taken by our population of students.

3. TEACHING PROGRAMING USING A MODEL-BASED APPROACH

In [22] three perspectives on the role of a programming language are described:

Instructing the computer The programming language is viewed as a high-level machine language. The focus is on aspects of program execution such as storage layout, control flow and persistence. In the following we refer to this perspective as coding.

Managing the program description The programming language is used for an overview and understanding of the entire program. The focus is on aspects such as visibility, encapsulation, modularity, separate compilation.

Conceptual modelling The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena.

When designing a programming course, one must balance the three perspectives; in a model-based programming course, by definition, conceptual modelling plays the most important role. In the course under consideration, the progression in the course is defined not by the syntactical structure of the programming language, as is usually the case [32], but by the complexity of specification models, i.e. class models and functional specifications of methods. Early in the course, examples, exercises and assignments address programming tasks described by simple specification models (one class only or two classes with a simple relationship and simple functional specifications); later in the course the programming activities are defined by more complex specification models (more classes with more advanced relations and more complex functional specifications).

The official ILO for the course is phrased as follows: After the course, the students must be able to apply fundamental constructs of a common programming language, identify and explain the architecture of simple programs, identify and explain the semantics of simple specification models, implement simple specification models in a common programming language, and apply standard classes for implementation tasks.

The evaluation of programming competences in this course is done by a 30 minute practical exam. For a description of how we measure the students‘ programming competences, see [6].

For a more detailed description of the model-based programming course design, see e.g. [4, 10, 5].

4. THE PROGRAMMING COURSE

The programming course under consideration spans the first half of CS1 at Aarhus University. The course runs for seven weeks, and after the course there is a practical lab examination with a binary pass/fail grading. The grading is based solely upon the behaviour in and result of the final examination; acceptable performance in weekly mandatory assignments during the course is a prerequisite for the final exam but does not count as part of the grading. There are approximately 350 students per year from a variety of study programmes, e.g. bio-technology, chemistry, computer science, mathematics, geology, nano science, economy, and multimedia. 40% of the students are majors in computer science; of course they continue with many more programming or programming related courses. For most of the remaining students, this is the only mandatory programming course in their curriculum, but some choose follow up courses as electives and some do have special follow up courses related to their field (e.g. multimedia programming or scientific computing).

Content
Getting started: Overview of fundamental concepts. Learning the IDE and other tools.
Learning the basics: Class, object, state, behaviour, control structures.
Conceptual framework and coding patterns: Control structures, data structures (collections), class relationship, patterns for implementing structure (class relationship)
Programming method: Stepwise improvement, schemes for implementing functionality.
Subject specific assignment: Practise on harder problems.
Practise: achieve routine in solving standard tasks.

Table 1: Course phases

The students are grouped in classes of approximately 20 students; typically there are 17-18 teams per year. Each class has its own teaching assistant (TA) who is typically a PhD student in computer science.

We adopt an incremental approach to programming education in which novices are provided with worked examples [40] and initially do very simple tasks and then gradually do more and more complex tasks, including design-in-the-small by adding new classes and methods to an already existing design. Table 1 gives an overview of the phases and content of the course.

For a more detailed discussion of the design of the course from a learning theoretic perspective, see [11].

5. RESEARCH QUESTIONS

As described in section 2, we forget things, and forgetting is more likely to take place when memory elements are not accessed and used. Programming fluency involves a lot of specific skills related to the programming language (syntax, semantics, and pragmatics), the development environment (editor, compiler, interpretation of error messages, and debugging), use of API, etc. The first category of skills, which we denote concrete programming competences, implies that programmers possess a great deal of fingertip knowledge about many specific, technical details and is therefore particular vulnerable with respect to being forgotten when not practised and applied,. Another category of programming skills and competences relate to problem solving and appli-

cation of patterns to solve recurring (types of) problems; we denote this abstract programming competences. The examination form ensures that these programing skills and competences have been present, but how long and how well do they last, and how easy is it to recall them? Our two hypotheses, which forms the basis for this research, are:

Forgetting The students have forgotten the concrete programming competences quickly after they have passed the course.

Learning It does not take much effort for the students to recall the concrete as well as more abstract programming competences.

The two hypotheses are operationalised into the following research questions:

RQ_1: Forgetting Have the students forgotten their concrete programming competences?

RQ_2: Learning Can the students with a limited effort recall their programming competences? And what are the challenges for recalling once learnt skills and competences?

6. RESEARCH DESIGN

This section describes the design of the research.

6.1 Participants

From the general cognitive theory, we expect that the students' programming competences are forgotten if not practised and applied. Thus, in order to test our hypotheses and answer our research questions, we need to identify a group of students who have not programmed since they passed the introductory programming course. This naturally rules out computer science students. As described in the introduction, many other students take programming classes, but this is not the case for students majoring in bio-technology.

Students from bio-technology take the introductory programming course in the third quarter of the first year. They have no other mandatory programming courses, and they do not practise programming as part of their studies. These students fulfil the overall requirement (they have not been programming for X months) and they are a group that can be addressed, since most of them still follow the same study program. There are currently 45 students in the bachelor program of bio-technology (14 in the first year, 17 in the second year, and 14 in the third year). This makes it difficult to do quantitative analyses (the number of students are too small in each group). Consequently, we have designed the research not with the focus of giving general, generalisable answers but rather as providing new insight and pointers to factors it might be interesting to investigate further.

Based on the research questions and the group of students that are accessible, we observe the students performing programming with a focus on the problems they encounter as they go along. We do this twice: A pre-test before the students get a chance to brush-up of their programming competences, and a post-test after the students have received a brush-up. Finally we interview the students in a semi-structured focus group interview.

Year	Months since prog. course	Male	Female	Programming since course
2007	27	1	0	course using MathLab
2008	15	0	4	none
2009	3	2	3	none

Table 2: The students participating in the experiment

6.2 Evaluation of Programming Competences

A key question is how we can evaluate the students programming competences? The exam of the course evaluates the learning goals of the course and consequently the programming competences the students should possess. We evaluate the students using two programming tests similar to the one used in the final exam of the introductory programming course. In [6] we argue that the exam actually measures the goals of the course. The pre-test can be seen in the appendix; the post-test is similar to the pre-test except for another cover story.

6.3 Rehearsing Programming Competences

The next question is what "limited effort" mean (RQ_2)? Shall the try to recall the students' programming competences through practise or through a general presentation of key concepts, techniques, and examples? And shall we provide some kind of assistance to recall their programming competences during the post-test?

Ideally we would like to "measure" the learning effort it takes a given student to be able to solve the task in the pre- and post-test. This is in practice impossible! As a compromise, we offer the students an overview of the central programming language constructs (basic statements, control structures, method, attribute, class, etc.) and central concepts such as association (one-to-many) and collections and how these are realised in the programming language (Java). Furthermore, we give the students one of two kinds of help when solving the post-test. In the final focus group interview we specifically address how the learning aids have helped the students.

6.4 Concrete experiment design

We invited all bio-technology students from the first, second, and third year to participate in the experiment (45 in total). 12 responded positively to our invitation, and 10 actually participated in the experiment. The students were not paid (apart from a dinner at the end), nor did they get course-credit for the experiment. The students had the characteristics described in table 2.

The experiment was conducted a late afternoon in a computer-lab (the same that was used for the lab-sessions during the course) and lasted 3 hours. The agenda for the experiment was as follows:

1. Welcome and introduction
2. Short repetition of use of the development environment (BlueJ [23])
3. Pre-test
4. Brush-up of programming competences

5. Post-test

6. Focus group interview

The welcome and introduction motivated the study and gave a general overview of the content of the afternoon. This part took 15 minutes.

The repetition of the development environment helped the students to remember how the IDE was designed and how to edit and compile programs. This was done via a few exercises the students had to solve — exercises from the textbook used when the students had the course [3]. This was done in order to have programming in focus, not the tool used for programming. The exercises included a small amount of actual programming (the students typed in some code that was provided, they did not develop the solution themselves). The students had therefore seen some Java code just before the pre-test. This part took 15–20 minutes (some students finished before others).

The pre-test was a standard assignment from a final exam. Four researchers observed the students (2-3 students per researcher). When the students got stuck, we noticed the problem and evaluated how the students tried to solve the problem. If the students had been stuck for a long period of time, we helped the students to move on and noted this help. The test lasted 30 minutes; same duration as the ordinary exam.

The brush-up of programming competences was done using some general slides from the introductory programming course. The slides describe general concepts (object, class, attribute, method, constructor, parameter, type, statement, selection, iteration, association and collection) and how these look in Java. The students could ask questions and discuss during the brush-up session. Nearly all of the students' questions were about specific details in Java. The students did not do practical programming during the brush-up session. This part of the experiment lasted one hour.

Also the post-test was a standard assignment from a final exam. In order to evaluate different aids, we divided the students into two groups: One group received a model solution for the pre-test, the other group received a general description of how to implement classes, associations and two algorithmic patterns (that typically occur in exam assignments): (1) in a collection of objects, find one that matches a given criteria, and (2) in a collection of objects, find all that matches a given criteria. The first help was very concrete; the second incorporated the idea of pattern-oriented instruction [28] which was emphasised in the ordinary course. As for the pre-test, four researchers observed the students and noted their difficulties. The post-test was also time-boxed to 30 minutes.

The focus group interview lasted 35 minutes and focused on the students difficulties, the difference between concrete programming competences and general competences, the effect of the intermediate learning task (the brush-up of programming competences), the influence of the aids provided, and general comments.

7. OBSERVATIONS AND ANALYSIS

This section describes and analyses the observations made during the experiment and the final focus group interview in order to answer the two research questions.

Month since prog. course	Last completed exercise	Problems
3	8	Did extremely well. Used `compareTo` instead of `equals` for checking if strings are equal.
3	6	Many problems with syntax like forgetting a method name.
3	5	Many syntactical problems.
3	none	Declared attributes in the constructor.
15	none	Methods without a signature. Confused about the value of a name-attribute and a reference to the given object.
15	none	Parameters for the values of the attributes in the `toString()` method.
27	none	Many syntactical problems. The `toString()` method was implemented by returning a string literal instead of values of variables.
3	4	Did fairly well. Wrote statements directly in the class without a surrounding method, but worked it out by himself.
15	3	Called a non-existing method (`gettoString()`).
15	none	Declared an attribute called `toString`. Declared attributes in the constructor.

Table 3: Each student's performance in the initial test

7.1 Forgetting

In this subsection we will look at RQ_1.

As expected, the concrete syntax was a major problem for almost all of the students. As one of the students noticed in the interview: *You quickly forget when to type a parenthesis or a semicolon - you can remember that it is important that they are put in the right place, but where that is ….* Another student expressed it the following way: *I had many problems in the first test. I could not remember how to write it — the class and the other stuff — I could remember that this class was a class and you can create objects from it, but in the code, I could not remember what to write and how to call. I could remember that you had to return something … but how it should be written and worked, I had totally forgotten .*

There was a difference between the students who took the course three month ago and the other students. All of the students had problems with the specific syntax, but the "younger" students (measured in time since they had the introductory course) had significantly less problems than the "older" students as can be seen from table 3. One of the

students (number 1) would actually have passed the test if it had been a real exam.

If we look more closely at the problems many students encountered in the pre-test, they include the following:

Attributes Many declared the attributes in the constructor and found it very difficult to initialise them.

Parameters Many found it difficult to declare parameters. It seemed like they had the idea of passing information through parameters but the concrete syntax was a problem.

Screen output vs. return value Many implemented the `toString()` method using a `System.out.println(...)`, and could not understand the error "missing return statement".

Programming process Many students gave up on a given question and left it unsolved even though it was required to solve the next question.

In general we conclude that the students had forgotten their specific programming competences. Only one student (who took the course three months ago) could solve more than very basic programming tasks. This student would have passed, had it been a real exam.

7.2 Learning

In this subsection we will look at RQ_2.

After the students had refreshed their programming competences, they performed significantly better as can be seen from table 4. If the post-test had been a real exam, seven of the ten students would have passed it!

In general, the aid that was provided helped the students. All of the students who had the model solution from the pre-test, performed well. In fact, they would all have passed had it been a real exam.

The students used the model solution in different ways. Some students started out on their own and just used the solution when they encountered a problem they could not solve by themselves. As one student said: *I did not use it for the first six questions ... there were something about ArrayList, how to write it, otherwise it was only in the end where you have to write a for-loop, I could not remember how to write that. I do understand the meaning and what it is, but I cannot remember how to write it.* Others used it more systematically: *I become a little stubborn when I get such one [a solution]. I want to do it by myself ... but I used it anyhow [for most of the test] because there were many things I could not remember .*

The performance of the students who got the general description was somewhat more diffuse. In general, they performed significantly better than in the pre-test, but not all would have passed had it been a real exam. Some students found it difficult to put the general solution to practice.

In general the *Refreshment of programming competences* phase in the experiment helped the students. As one student said *I think it helped me a lot — the PowerPoint show — because I had completely forgotten all. I actually think I had forgot that there should be a list if it wasn't told.*

In the *Refreshment of programming competences* phase, many students had good and in-depth questions using correct terminology for programming concepts. We see this as

Month since prog. course	Last completed exercise	type of help	Problems
3	9	G	None.
3	9	S	Forgot to include statements in { }.
3	9	G	None.
3	2	G	Wrote literals instead of identifiers in the parameter list of the constructor.
15	7	S	None.
15	8	S	None.
27	5	G	Forgot to import `java.util.*`.
3	9	S	None.
15	8	S	None.
15	4	G	Instead of type-identifier pairs in the parameter list, she wrote identifier-identifier pairs where the first identifier was the attribute and the second was the parameter.

Table 4: Each student's performance in the second test. S referees to a solution of the initial test, G to a general description of how to implement different structures.

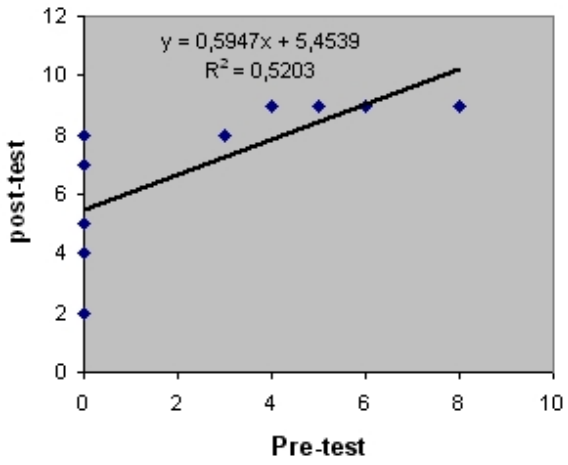


Figure 2: Number of completed exercises in the pre- and post test

an additional indicator that the students may have forgotten the syntax but the more conceptual content and general competences and skills are more easy to recall.

The design of this study was to use a qualitative research approach, where we observed what the students did, what problems they encountered and abstracted these findings. An alternative way to address the research question (RQ_2) could be to statistically check if the students performed better after the intervention. Figure 2 plots the students number of completed exercises in the pre- and post-test. If we analyse the data using linear regression [27], we can observe that there is a reasonably strog correlation between the observations ($R^2 = 0.52$), and that the line is well above the diagonal. This supports the conclusion that the intervention indeed helped the students recall their programming competences. However, as noted initially, the number of students in this study was only ten.

In general, we conclude that the students with the help they got (one hour of lecturing plus help during the test) could recall their programming competences. Consequently, we conclude that it is possible with a limited effort for most of the students in this study to recall general as well as more specific programming competences and skills.

The other part of RQ_2 "What are the challenges for recalling once learnt skills and competences?" is more difficult to answer.

8. FUTURE WORK

In this study only ten students from one study program participated. It will be interesting to expand the findings from this research by involving more students from more study programs. Fortunately, students from several other study programs who do not receive further programming instruction, have taken the course.

Programming is being taught in many different ways, and there are many different ways of phrasing the intended learning outcome of introductory programming courses. In order to obtain more reliable and generalisable results, it would be interesting to include more universities and colleges in the research and thus aim for a multi-institutional (and multi-national) study. As [36] argues, a *multi-national, multi-institutional context, defines a new interface between computer science education research and computer science education practice — hopefully bringing them closer together* (p. S4E-16).

9. CONCLUSION

We have conducted a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming competence long time after the educational activity has taken place.

In the pre-test, all students struggled with syntax issues, but the younger students (measured in time since they had the introductory course) had significantly less problems than the older students.

Our qualitative study indicates, not surprisingly, that syntactical issues in general hinder immediate programming productivity, but more interestingly it also indicate that a tiny retraining activity and simple guidelines is enough to recall general as well as more specific programming competences and overcome syntactical issues.

10. REFERENCES

[1] C. Alphonce and P. Ventura. Object orientation in cs1-cs2 by design. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 70–74. ACM Press, 2002.

[2] M. Anderson, R. Bjork, and E. Bjork. Remembering can cause forgetting: Retrieval dynamics in long-term memory. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 20(5):1063–1087, 1994.

[3] D. J. Barnes and M. Kï£¡lling. *Objects First With Java: A Practical Introduction Using Bluej*. Pearson, Essex, United Kingdom, 3rd edition, 2006.

[4] J. Bennedsen. *Teaching and learning introductory programming - a model-based approach*. PhD thesis, University of Oslo, Norway, department of Computer Science, 2008. accessed May, 2009.

[5] J. Bennedsen and M. Capersen. Model-driven programming. In J. Bennedsen, M. Caspersen, and M. Kï£¡lling, editors, *Reflections on the Teaching of Programming*, pages 116–129. Springer-Verlag, Berlin, Germany, 2008.

[6] J. Bennedsen and M. E. Caspersen. Assessing process and product ï£¡ a practical lab exam for an introductory programming course. *ITALICS, Innovation in Teaching and Learning in Information and Computer Sciences*, 6(4):183–202, 2007.

[7] J. Bennedsen and M. E. Caspersen. Optimists have more fun, but do they learn better? ï£¡ on the influence of emotional and social factors on learning introductory computer science. *Journal of Computer Science Education*, 18(1):1–16, 2008.

[8] A. J. Biamonte. Predicting success in programmer training. In *SIGCPR '64: Proceedings of the second SIGCPR conference on Computer personnel research*, pages 9–12, New York, NY, USA, 1964. ACM Press.

[9] R. Bjork. Retrieval practice and the maintenance of knowledge. In M. Gruneberg, P. Morris, and R. Sykes, editors, *Practical aspects of memory: Current research and issues*, volume 1, pages 396–401. Chichester, England, 1988.

[10] M. E. Caspersen. *Educating Novices in the Skills of Programming*. PhD thesis, Aarhus University, Department of Computer Science, 2007. accessed May 2009.

[11] 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, New York, NY, USA, 2007. ACM.

[12] G. Cooper and J. Sweller. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79(4):347–362, 1987.

[13] H. Cooper, J. C. Valentine, K. Charlton, and A. Melson. The Effects of Modified School Calendars on Student Achievement and on School and Community Attitudes. *Review of Educational Research*, 73(1):1–52, 2003.

[14] H. Ebbinghaus. *ï£¡ber das Gedï£¡chtnis*. Teachers College, Columbia University, New York, New York, United States, 1885.

[15] H. Ebbinghaus. Memory: A contribution to experimental psychology. `http://psychclassics.yorku.ca/Ebbinghaus/index.htm`, 1885. Translated from German by Henry A. Ruger and Clara E. Bussenius (1913). Last accessed May 14, 2009.

[16] J. T. Gorgone, G. B. Davis, J. S. Valacich, H. Topi, D. L. Feinstein, and J. Herbert E. Longenecker. Is 2002 - model curriculum and guidelines for undergraduate degree programs in information systems. Retrieved June 2009, 2002.

[17] J. T. Gorgone, P. Gray, E. A. Stohr, J. S. Valacich, and R. T. Wigand. Msis 2006: model curriculum and guidelines for graduate degree programs in information systems. *SIGCSE Bull.*, 38(2):121–196, 2006.

[18] M. Guzdial and A. Forte. Design process for a non-majors computing course. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 361–365, New York, NY, USA, 2005. ACM.

[19] T. B. Hilburn. A top-down approach to teaching an introductory computer science course. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, 25(1):58–62, 1993.

[20] E. Howe, M. Thornton, and B. W. Weide. Components-first approaches to cs1/cs2: principles and practice. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 291–295. ACM Press, 2004.

[21] J. Jain, I. James H. Cross, and D. Hendrix. Qualitative comparison of systems facilitating data structure visualization. In *ACM-SE 43: Proceedings of the fourty-third annual Southeast regional conference*, pages 309–314, Kennesaw, Georgia, 2005. ACM Press.

[22] J. L. Knudsen and O. L. Madsen. Teaching object-oriented programming is more than teaching object-oriented programming languages. In S. Gjessing and K. Nygaard, editors, *ECOOP '88 European Conference on Object-Oriented Programming*, pages 21–40, Berlin, Germany, August 15-17, 1988 1988. Springer-Verlag.

[23] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[24] R. B.-B. Levy, M. Ben-Ari, and P. A. Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40(1):1 – 15, 2003.

[25] E. Loftus. *Memory: surprising new insights into how we remember and why we forget*. Addison-Wesley, Reading, Massachusetts, United States, 1980.

[26] S. Matzko and T. A. Davis. Teaching cs1 with graphics and c. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 168–172, New York, NY, USA, 2006. ACM Press.

[27] D. C. Montgomery and E. A. Peck. *Introduction to linear regression analysis*. John Wiley, New York, NY, USA, 1982.

[28] O. Muller, D. Ginat, and B. Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.*, 39(3):151–155, 2007.

[29] R. E. Pattis. The ï£¡procedures earlyï£¡ approach in cs 1: a heresy. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 122–126. ACM Press, 1993.

[30] M. M. Reek. A top-down approach to teaching programming. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, pages 6–9, New York, NY, USA, 1995. ACM Press.

[31] S. Reges. Conservatively radical java in cs1. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 85–89. ACM Press, 2000.

[32] A. Robins, J. Rountree, and N. Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.

[33] D. Schacter. The seven sins of memory: Insights from psychology and cognitive neuroscience. *American Psychologist*, 54:182–203, 1999.

[34] A. Schmolitzky. "objects first, interfaces next" or interfaces before inheritance. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 64–67. ACM Press, 2004.

[35] R. Shackelford, J. H. C. II, G. Davies, J. Impagliazzo, R. Kamali, R. LeBlanc, B. Lunt, A. McGettrick, R. Sloan, and H. Topi. The overview report. Accessed June 2009, 2006.

[36] B. Simon, R. Lister, and S. Fincher. Multi-institutional computer science educational research: A review of recent studies of novice

understanding. In *in Proceedings of the 36th Annual Frontiers in Education Conference*, pages SE412–17, October 2006.

[37] D. Soldan, J. L. Hughes, J. Impagliazzo, A. McGettrick, V. P. Nelson, P. K. Srimani, and M. D. Theys. Computer engineering 2004 - curriculum guidelines for undergraduate degree programs in computer engineering. Accessed June 2009, 2004.

[38] J. Stasko, A. Badre, and C. Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 61–66, Amsterdam, The Netherlands, 1993. ACM.

[39] L. A. Stein. What we swept under the rug: Radically rethinking cs1. *Computer Science Education*, 8(2):118–129, 1998.

[40] J. Sweller and G. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985.

[41] D. W. Valentine. Cs educational research: a meta-analysis of sigcse technical symposium proceedings. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 255–259, New York, NY, USA, 2004. ACM.

[42] S. Wiedenbeck. Factors affecting the success of non-majors in learning to program. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 13–24, New York, NY, USA, 2005. ACM Press.

[43] J. Wixted and E. Ebbesen. Genuine power curves in forgetting: A quantitative analysis of individual subject forgetting functions. *Memory and Cognition*, 25:731–739, 1997.

[44] U. Wolz and E. Koffman. Interactivity in cs1 & cs2: bringing back the fun stuff with java. In *CCSC '00: Proceedings of the fifth Annual Consortium for Computing Sciences in Colleges CCSC: Northeastern conference*, pages 1–3. Consortium for Computing Sciences in Colleges, 2000.

[45] R. Woodworth. *Experimental Psychology*. Henry Holt, New York, United States, 1938.

Aarhus University
Dept of computer science

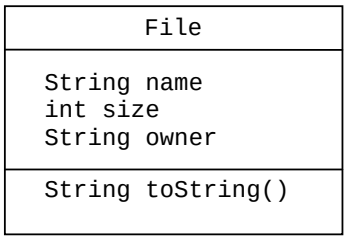
APPENDIX

Pre-test
June 3rd 2009

Experiment 1, (Pre-test) Post-test is similar except that another domain and slightly modified methods are used

1. Create a class, *File*, representing a file; the class *File* is specified in the UML-diagram to the right. The three attributes must be initialized in a constructor (using parameters of appropriate type). The method *toString* returns a string-representation of a file, e.g.

   ```
   "testexercises.doc, 267 kb, mec"
   ```

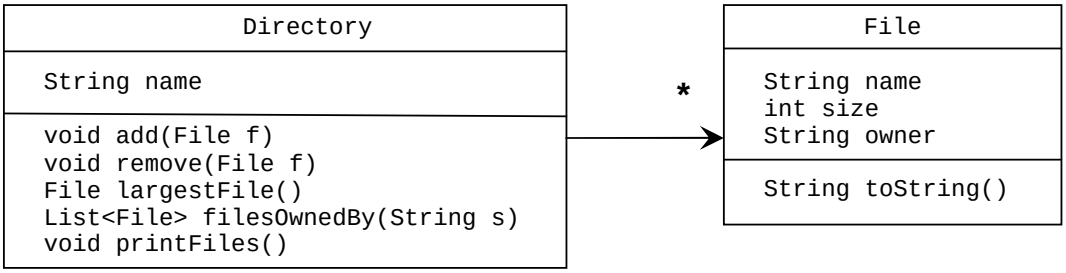


2. Create a *Driver*-class containing an *exam*-method. The method must be static, have return type void and no parameters.

3. Create three *File*-objects, using object references *f*1, *f*2 and *f*3, in the *exam*-method and print out these using the *toString*-method.

Call the observer and demonstrate what you have made so far.

4. Create a new class, *Directory*, representing a directory in a file-system. The class *Directory*, and its relation to the *File* class, is specified in the following UML-diagram:



5. Program the methods *add* and *remove* who respectively adds and removes the *File*-object *f* to/from the Directory-object.

6. Create an object of type *Directory* in the *exam*-method in the *Driver*-class and associate the already created *File*-objects to this object.

7. Program the method *largestFile*. The method returns the larges file in the directory (it can be assumed that the directory is not empty; if two or more files have the same size it is subordinate which file thet is returned). Extend the *File*-class with the necessary get-methods.

8. Use the method *largestFile* in the *exam*-method in the *Driver*-class tio print out information on the largest file in a directory.

Call the observer and demonstrate what you have made so far.

9. Program the method *filesOwnedBy*. The method must return a list of files owned by *s*. Extend the *File*-class with the necessary get-methods.

10. Program the method *printFiles*. The method prints out a list of all files in a directory arranged by size.

Call the observer and demonstrate your final solution.