Computer Science Education
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ncse20

Persistence of elementary programming skills

Jens Bennedsen [a] & Michael E. Caspersen [b]

[a] Aarhus University School of Engineering, Finlandsgade 22, DK-8200 Aarhus N, Denmark

[b] Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark

Version of record first published: 25 Jun 2012

To cite this article: Jens Bennedsen & Michael E. Caspersen (2012): Persistence of elementary programming skills, Computer Science Education, 22:2, 81-107

To link to this article: http://dx.doi.org/10.1080/08993408.2012.692911

Computer Science Education
Vol. 22, No. 2, June 2012, 81–107

Routledge
Taylor & Francis Group

Persistence of elementary programming skills

Jens Bennedsen[a]* and Michael E. Caspersen[b]

[a]*Aarhus University School of Engineering, Finlandsgade 22, DK-8200 Aarhus N, Denmark;* [b]*Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark*

(*Received 30 April 2012; final version received 8 May 2012*)

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 10 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.

Keywords: CS1; object-oriented programming; remembering

1. Introduction

There is a long-lasting and intense interest in programming education development and research; however, most of the research concerns teaching methods, educational technology and student understanding/misconceptions. Very little, if any, research has investigated the long-term effect of programming education.

There is plenty of research in introductory programming education, but in general this research focuses on student performance or behaviour based on data collected during or immediately after the course with a focus on e.g. learning outcome, learning obstacles, misconceptions and indicators of success. We are interested in the persistence of programming

*Corresponding author. Email: jbb@iha.dk

ISSN 0899-3408 print/ISSN 1744-5175 online

http://dx.doi.org/10.1080/08993408.2012.692911
http://www.tandfonline.com

skills, i.e. how well students can remember skills once learnt and not practiced for some time.

The main forums for programming education development and research are the annual American-based conference on computer science education (Technical Symposium on Computer Science Education, SIGCSE) and its European counterpart, Innovation and Technology in Computer Science Education (ITiCSE). SIGCSE was held for the 43rd time in 2012, and ITiCSE will be held for the 17th time in 2012. Most of the publications within the field of computer science education research are ''practitioner reports'' (Carbone & Kaasbll, 1998; Fincher & Petre, 2004; Holmboe, 2005), but a current change is marked by more research-based publications. ACM is hosting a conference aimed specifically at Computer Science Education Research (International Computing Education Research Workshop, ICER), to be held for the eighth time in 2012 as well as the Baltic conference series on computer science education (Koli Calling) to be held for the 12th time in 2012.

In the late sixties and early seventies, a special interest in programming as a domain to study cognition emerged. Many consider the book, ''The Psychology of Computer Programming'', by Gerald M. Weinberg in 1971, to be the first book within the field (it was republished in 1998 in a silver anniversary edition [Weinberg, 1998]). In the books' preface he writes, ''This book has only one major purpose to trigger the beginning of a new field of study: computer programming as a human activity'' (p. vii). In the seventies, programming cognition became an active field with a focus on how expert programmers differ from novices.

Teaching methods, materials and educational technology (see e.g. Kumar 2004; Levy, Ben-Ari, & Uronen 2003; Malmi, et al. 2004) have been developed with the aim of improving students learning of computer science and especially programming. Many innovations have their out-spring in academia and many have a double purpose: technical as well as educational research. Studies that evaluate educational technology (see e.g. Jain et al., 2005; Levy et al., 2003; Stasko, Badre, & Lewis, 1993), studies that evaluate the usefulness of the competences learnt by the students when they enter their first job (Begel & Simon, 2008) as well as studies in what students find most problematic ((Butler & Morgan, 2007; Milne & Rowe, 2002; Schulte & Bennedsen, 2006) are examples of such studies). 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 (this is naturally not true with the studies about the usefulness of competences for students in their first job). Evaluating the impact immediately after the course is of course both interesting and relevant, but in general the goals of our teaching are 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 (Guzdial & Forte, 2005). There are furthermore a growing number of study-programs that combines computer science with something else, e.g. business, media and healthcare. Many introductory computing courses have programming as a core activity and learning goal, and for good reasons since programmability is the defining characteristic of the (digital) computer. This is also echoed in the ACM/IEEE curriculum recommendations *the programming-first model is likely to remain dominant for the foreseeable future (p. 24)*. 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 (Shackelford et al., 2006), from Information Systems (Gorgone et al., 2002) to Computer Engineering (Soldan et al., 2004).In e.g. "the model curriculum and guidelines for graduate degree programs in information systems" (Gorgone, Gray, Stohr, Valacich, & Wigand, 2006), 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 (Bjork, 1988). By fitting data from several experiments in cognitive psychology, Woodworth (1938) 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 this research, we are particularly interested in studying the persistence of elementary programming skills and competencies achieved some time ago and that have not been applied in the meantime. Computer science (CS) majors regularly practice programming; consequently, our

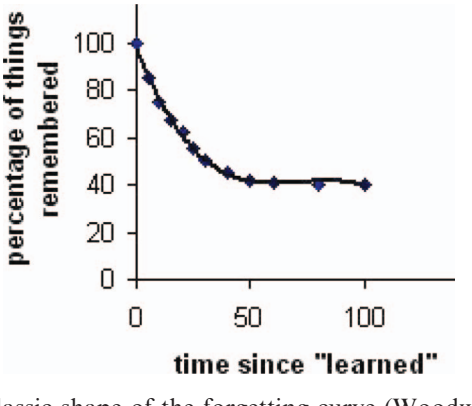


Figure 1. Classic shape of the forgetting curve (Woodworth, 1938).

research focuses on non-CS majors. Our subjects have gained their programming competences in an introductory object-oriented and model-based programming course. 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 10 students who have taken an introductory object-oriented programming course 3, 15 or 27 months prior to our test. None of the students, who are majors in bio-technology, have been exposed to programming in the intervening period.

The remaining part of the article is organised as follows: Section 2 describes related work primarily in cognitive psychology. In Section 3, we describe the instructional design of the introductory programming course. Section 4 presents our hypotheses and research questions, and Section 5 presents our research design. In Section 6 we describe and analyse our observations. Potential future work is described in Section 7, and Section 8 is the conclusion.

2. Related work

When practicing programming, many different elements are in play; consequently, many areas are related to maintaining programming skills or, as the other side of the coin, forgetting how to program. For example, Kim and Lerch (1997) view programming as search in three problem spaces: rule, instance and representation, and Caspersen and Kolling (2009) view programming as navigation in a three-dimensional space of refinement, extension and restructuring. This section focuses on two related areas: work in the area of human memory and work in the area of remembering programming competences.

2.1. Human memory

Human memory is fallible. The fact that people gradually forget was first documented by Ebbinghaus (1885a) 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, [Ebbinghaus, 1885b]). Ebbinghaus found that a complex logarithmic function described his data. Later, it has been shown (Wixted & Ebbesen, 1997) that a power function $y = \alpha\ t^{\beta}$ better describes the relation between time and remembering. The values of α and β rely upon the actual person and the ''thing'' to remember. The findings of Ebbinghaus – that forgetting occurs rapidly at first and then slows down – have been confirmed by later studies in laboratories (Wickelgren, 1972). There are some conflicting evidence about whether forgetting occurs at all in very long-tern memory (Bahrick, 1984; Bahrick, Bahrick, & Wittlinger, 1975; Squire, 1999) – very long-term meaning more than 10 years. In this study we focus on long-term memory only.

Memory links the past with the present. Tulving (1985) describes two different forms of recognition: remember and know. Gardiner and Java (1991) describe it this way: Recognition can be accompanied by either conscious recollection of some specific experience or feelings of familiarity without any recollective experience. Recognition memory with and without recollective experience can be measured by ''remember'' and ''know'' responses. A ''remember'' response indicates that seeing the word in the test list brings back to mind some specific recollection of what was experienced when the word appeared in the study list. A ''know'' response indicates that seeing the word in the test list brings to mind feelings of familiarity, without any recollective experience. (p. 617). Recalling programming competence is expected to be a ''remember'' experience. Quite a few independent variables have been found to influence ''remember'' responses but not ''know'' responses among others the number of rehearsals (Macken & Hampson, 1991). However, it has been shown that rate of forgetting is not influenced by whether it is measured for easy or difficult items (Slamecka & McElree, 1983).

Models of the human cognitive architecture recognise two memory components: working memory and long-term memory (Newell, Rosenbloom, & Laird, 1989) (Figure 2). All human learning and activities rely on these two components and the partnership they share. As its name implies, working memory is the active partner (as you read this and think about its relevance to the paper, it is your working memory that does the processing). While in learning mode, new information from the environment is processed in working memory to form knowledge structures called schemas, which are stored in long-term memory. Schemas are memory structures that permit us to treat a large number of information elements as if they are a single element. New information entering working memory must be integrated into pre-existing schemas in long-term memory. For this to take place, relevant schemas in long-term memory must be activated and decoded into working memory, where integration takes place. The result is an encoding of extended schemas stored in long-term memory. Learning nonsense syllables will not be

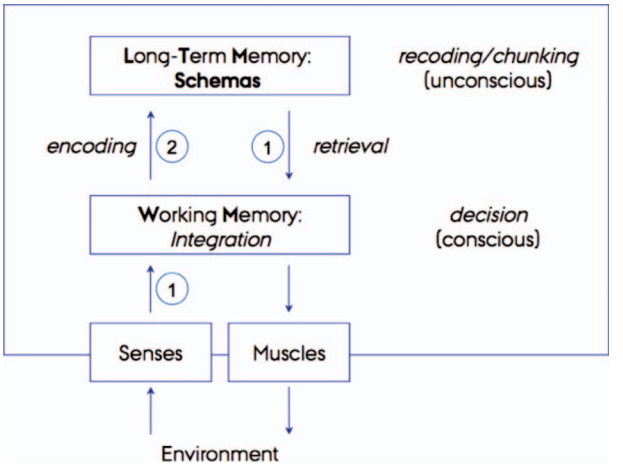


Figure 2. A model of the human cognitive architecture.

integrated into the long term memory, but forgetting also happens for schemas in the long-term memory (Anderson, Bjork, & Bjork, 1994).

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, [Ebbinghaus, 1885b]).

Re-learning affects forgetting. As Schacter (1999) notices 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 (1980) has 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 et al. (1994) 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 associated to the same cue or cues guiding retrieval – that may be put in greater jeopardy of being forgotten. (p. 1063). According to Anderson et al. (1994), the reason for this is three assumptions on how the memory works:

(1) *The competition assumption*: Memories associated to a common cue compete for access to conscious recall when that cue is presented,

(2) *Strength dependence assumption*: A cued recall of a memory will decrease as a function of increases in the strength of its competitors,
(3) *Retrieval-based learning assumption*: Recall of a memory enhances subsequent recall of that memory.

Based on the knowledge of forgetting, several studies have been made in order to evaluate the effect of students' breaks (Cooper, Valentine, Charlton, & Melson, 2003). We think that most teachers know that after a long summer vacation, it is much harder for students to e.g. program again. 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 (Cooper & Sweller, 1987). 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 programming*

We are not aware of research in programming education similar to the one reported here where we investigate the persistence of elementary programming skills, i.e. how well students can remember or recall programming skills learnt some time ago and not practiced in the meantime.

There is a lot of empirical research in introductory programming education, but in general this research focuses on learning outcome measured during the course or immediately after the course has finished with a focus on e.g. learning outcome, learning obstacles, misconceptions and indicators of success. In particular, a substantial amount of research has been conducted to identify general variables that predict the success of students aiming for a university degree within computer science – especially focusing at predicting the success of learning programming. Bennedsen (2008) presents en extensive list of such studies.

In the sixties, a lot of work on creating and validating psychological test to select programmers were performed. Much of the work of the Special Interest Group in Computer Personnel Research (SIGCPR) was about psychological tests for the selection of computing staff. Back then there were not many people educated in the field, but the industry had a huge demand for manpower. In 1966, the number of programmers and system analysts ranged from 170,000 to 200,000; and the number was expected to rise to 400,000 in 1970 (Dickmann & Lockwood, 1966).

Simpson (1973) published a bibliography in 1973 containing 152 publications describing tests for programming ability.

In conclusion, studies of the long-term effects of learning how to program are very rare. In general, the empirical research in the area of computer science education research on introductory programming focuses on the effect obtained immediately after a programming course.

3. Teaching programming

The purpose of this section is to give a description of the philosophy, beliefs, values and perspectives behind the programming course through which our subjects have learned their programming skills. Globally, there is a lot of variation in introductory programming courses, and in particular in the philosophy, beliefs, values and, perspectives of people teaching these courses. If we take the vast majority of textbooks as indicators of typical introductory programming courses, the course in play is atypical. In order to appreciate our research and enable replication, we provide a thorough description of the course and its philosophy. We start with a description of the variation in approaches to introductory programming courses; following this, we sketch a model-based approach to teaching programming; we conclude the section with specific information about the programming course in play.

3.1. Variation in approaches to teaching programming

Many approaches to introductory programming education have been proposed including a procedures early approach (Pattis, 1993), a top-down approach (Hilburn, 1993; Reek, 1995), a graphics approach (Matzko & Davis, 2006). Even within introductory object-oriented programming, many different approaches exist: objects early (Alphonce & Ventura, 2002), interfaces early (Schmolitzky, 2004), GUIs early (Wolz & Koffman, 2000), concurrency early (Reges, 2000), events early (Stein, 1998), components early (Howe, Thornton, & Weide, 2004), etc.

All of these articles about introductory programming education describe different (groups of) people's approaches. Despite a common curriculum (Engel & Roberts, 2001), many different interpretations of the curriculum exist. This could be the reason why so many different approaches to teaching programming coexist and are promoted as being the best. The authors argue that a certain approach is better than others based on the (often implicit) assumption that certain learning outcomes should be promoted.

To give a more thorough understanding of the concrete expected programming competences, in the following subsections we will describe the philosophy, beliefs, values and perspectives of the introductory object-oriented programming course taken by the students in this research.

3.2. A model-based approach to teaching programming

Knudsen and Madsen (1988) describe three perspectives on the role of a programming language:

(1) *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.
(2) *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 and separate compilation.
(3) *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. The progression of the course in play is defined not by the syntactical structure of the programming language, as is usually the case (Robins, Rountree, & Rountree, 2003), but by the complexity of specification models, i.e. class models and (informal) 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).

We adopt an incremental approach to programming education in which novices are provided with worked examples (Sweller & Cooper, 1985) 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.

In particular, we emphasise *algorithmic patterns* as one of the key concepts. Like design patterns (Gamma, Helm, Johnson, & Vlissides, 1995), algorithmic patterns describe solutions to common problems, not at the software design level but at the algorithmic level of programming. Sweeping through a data set is a standard algorithmic pattern, so is searching, divide-and-conquer and backtracking to name a few (Sedgewick & Schidlowsky, 1998).

We emphasise our pattern-oriented instruction because we believe it supports long-term learning and thus has implications for the present research.

In a model-based approach, algorithmic/procedural aspects and structural/organisational aspects of object-oriented programming go nicely hand-in-hand, and we emphasise both aspects through patterns. For example, the algorithmic/procedural aspect of iterating through a set or list is partially captured by two algorithmic patterns, `findOne` and `findAll`, and the structural/organisational aspect of zero-to-many relations between classes/objects is captured in the elementary design pattern `*-Association`.

As mentioned, these aspects go nicely hand-in-hand. A `*-Association` in a class model, implemented by a set or list, invite methods with a select-like functionality like `findOne` and `findAll` to compute associated objects satisfying a certain predicate. For example, for an Account-Transaction association, it could be relevant to find all transactions within a certain timeframe or all transactions of at least a certain amount. In the case of a Playlist-Track association, it could be all tracks with a certain artist, the most popular track, or all tracks of a certain genre.

Listing 1 shows the two algorithmic patterns for finding one or all associated objects satisfying a certain criteria of a `*-Association`.

Through several similar examples (same structure but different cover story), we urge the students to inductively identify algorithmic patterns for similar standard problems; occasionally, we deductively provide a pattern up-front and ask the student to apply the pattern to a number of similar problems. The choice of approach (inductive or deductive) depends on the audience and the situation.

Listing 1 Patterns for implementing `findOne` and `findAll`

```
class B{ . . . }

class A{
  . . .
  private List < B > bs;
  public B findOneX() {
    B res = bs.get(0);
    for (B b: bs) {
      if (/*b is a better X than res*/) {
        res = b;
    } }
    return res;
  }
  public List < B > findAllX() {
    List < B > res = new ArrayList < B > ();
    for (B b: bs) {
      if (/*b satisfies criteria X*/) {
```

```
        res.add(b);
    } }
    return res;
  }
  ...
}
```

In Experiment 1 (pre-test) in the appendix, method `largestFile` is an instance of `findOne` and `filesOwnedBy` is an instance of `findAll`. In Experiment 2 (post-test), method `mostExpensive` is also an instance of `findOne`, but there is no strict instance of `findAll`; instead, method `totalValueOfCars` requires a simple sweep of the dealer's list of cars.

Worked examples that the students complete before they embark on similar problems, and faded guidance help focus on the essential aspects of a programming task, and the specific details of the programming language becomes means to an end instead of a goal in itself.

For a more detailed description of a model-based programming course design, (Bennedsen, 2008; Bennedsen & Caspersen, 2004, 2008; Caspersen, 2007; Caspersen & Bennedsen, 2007). Caspersen and Bennedsen (2007) discuss and argue for the design from a learning theoretic perspective. A more elaborate and varied set of reflections on the teaching of programming in general and object-oriented programming in particular is available in Bennedsen, Caspersen, and Kölling (2008).

3.3. Specifics about the programming course in play

In this section, we describe the programming course taken by the students in this research. It is a model-based programming course which 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. For a thorough description of how we measure the students' programming competences, see Bennedsen and Caspersen (2007).

The official intended learning outcomes (ILOs) 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.

There are approximately 400 students per year from a variety of study programmes, e.g. bio-technology, chemistry, computer science,

IT, mathematics, geology, nano-science, economy and multimedia. Thirty to forty per cent of the students are majors in computer science; of course, they continue with many more programming or programming-related courses. For many 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).

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

Table 1 gives an overview of the phases and content of the course.

After seven weeks, the students are able to implement simple class models with 4–5 classes connected by the standard relations of aggregation (composition) and association. A majority of the students are able to implement `findOne` functionality by letting the associated class (B) implement the interface `Comparable`; with a suitable implementation of `Comparable`, the body of method `findOne` can be implemented as a simple call of the `min` or `max` method of class `Collections`.

In the last few weeks before the exam, the students solve a great deal of problems similar to the one in the Appendix. For the final exam, the students must be able to solve a similar assignment in 30 min. Three weeks before the exam, it takes an average student a couple of hours to solve such a problem, and they think they will never be able to pass the exam (and they complain about the ''stupid'' 30-min constraint); but they will (the failure rate is approximately 10%).

There is a good reason for the 30-min requirement. It forces the students to practice, and practice makes them learn the routines and the basic craftsmanship of programming, and that is the whole purpose. The power law of practice (Newell, Rosenbloom, & Anderson, 1981) tells us that, depending on the degree of learning, it takes only a certain amount of practice (trials) to get below the 30-min time limit, but the practice reinforces procedural solve-this-kind-of-problem-schemas in long-term

Table 1. Course phases.

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.

memory and thus reinforces learning. It is our hypothesis that this will help the students to maintain their basic programming skills for many years.

4. Research questions

As described in Section 2, we forget things, and forgetting is more likely to take place when elements in long-term memory 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 particularly vulnerable with respect to being forgotten when not practised and applied. Another category of programming skills and competences relate to problem solving and application of patterns to solve recurring (types of) problems; we denote this abstract programming competences. The examination form ensures that these programming 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 form the basis for this research, are:

(1) H_1 (forgetting): The students have forgotten the concrete programming competences quickly after they have passed the course.
(2) H_2 (recalling): 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 (H_1 into RQ_1 and H_2 into $RQ_{2.1}$ and $RQ_{2.2}$):

(1) RQ_1: To which extend have the students forgotten their concrete programming competences?
(2) $RQ_{2.1}$: Can the students with a limited effort recall their programming competences?
(3) $RQ_{2.2}$: What are the challenges for recalling once learnt skills and competences?

5. Research design

The description of the research design is broken down into four parts: we describe the candidate participants, the way we evaluate programming competence, the way we facilitate the students in rehearsing their

programming competence and, finally, concrete details about the organisation of the experiment and our data collection.

5.1. Candidate 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 is 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 that might be interesting to investigate further.

5.2. Evaluation of programming competences

Comparing studies done in different courses are difficult. To properly evaluate the long-time learning effect of a programming course, we must take as starting point the ILOs of the course in play. Whether the ILO focuses on special features of the programming language, the process of program development, problem decomposition or something else, has an impact on what programming competences the subjects need to have remembrance of. The concrete ILOs of the course the subjects had taken were described in Section 3.3.

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 Bennedsen and Caspersen (2007), we argue that the exam

actually measures the goals of the course. The tests can be seen in the appendix.

5.3. Rehearsing programming competences

The next question is what ''limited effort'' means (see $RQ_{2.1}$)? Shall we 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 specific programming competences during the post-test (e.g. syntax)?

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, but in practice this is 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.

5.4. Concrete experiment design

We invited all bio-technology students from the first, second and third year to participate in the experiment (45 in total). Twelve 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.

We observed the students performing programming with a focus on the problems they encountered as they went along. We did this twice: At a pre-test before the students got a chance to brush-up their programming competences, and at a post-test after the students had received the brush-up. Finally, we interviewed the students in a semi-structured focus group interview.

Table 2. The students participating in the experiment.

Year	Months since programming course	Male	Female	Programming since course
2007	27	1	0	Course using MathLab
2008	15	0	4	None
2009	3	2	3	None

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

(1) Welcome and introduction
(2) Short repetition of use of the development environment (BlueJ [Kölling, Quig, Patterson, & Rosenberg, 2003])
(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 min.

The repetition of the development environment helped the students to remember how the interactive development environment (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 took the course (Barnes & Kolling, 2006). 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 min (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 min; 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 1 h.

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 (Muller, Ginat, & Haberman, 2007), 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 min.

The focus group interview lasted 35 min 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.

6. 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 (three) research questions.

6.1. 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 this 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 months 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:

(1) *Attributes*: Many declared the attributes in the constructor and found it very difficult to initialise them.
(2) *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.

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

Month since programming 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.

(3) *Screen output vs. return value*: Many implemented the `toString()` method using a `System.out.println(...)`, and could not understand the error "missing return statement".
(4) *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.

6.2. *Learning*

In this subsection, we will look at $RQ_{2.1}$ and $RQ_{2.2}$.

After the students had refreshed their programming competences, they performed significantly better as can be seen by comparing the two

rightmost columns from Tables 3 and 4: All of the students completed a larger part of the test and the problems they encountered were simpler. If the post-test had been a real exam, 7 of the 10 students would have passed it.

The design of this study was to use a qualitative research approach, where we observed what the students did, what problems they encountered, and then abstracted these findings. An alternative way to address the research question ($RQ_{2.1}$) could be to statistically check if the students performed better after the intervention. Figure 3 plots the students' number of completed exercises in the pre- and post-test. If we analyse the data using linear regression (Montgomery & Peck, 1982), we can observe that there is a reasonably strong 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 10.

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.

$RQ_{2.2}$ "What are the challenges for recalling once learnt skills and competences?" is more difficult to answer. In general, the aid that was

Table 4. Each student's performance in the second test.

Month since programming course	Type of help	Last completed exercise	Problems
3	G	9	None.
3	S	9	Forgot to include statements in {}.
3	G	9	None.
3	G	2	Wrote literals instead of identifiers in the parameter list of the constructor.
15	S	7	None.
15	S	8	None.
27	G	5	Forgot to import java.util.*.
3	S	9	None.
15	S	8	None.
15	G	4	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.

Note: S referees to a solution of the initial test, G to a general description of how to implement different structures.

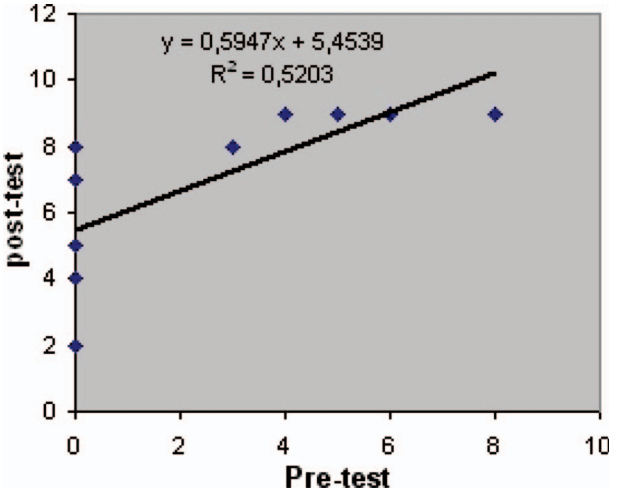


Figure 3. Number of completed exercises in the pre- and post-test.

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

7. Future work

In this study, only 10 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 Simon, Lister, & Fincher (2006) 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).

8. Conclusion

We have conducted a qualitative investigation of sustainability of programming competence by studying the effect of recalling programming skills and competence learnt some time ago and not practiced in the meantime.

In the pre-test, all students struggled with syntax issues, but the younger students (measured in time since they took 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. More interestingly, it also indicates that a tiny retraining activity and simple guidelines are enough to recall general as well as more specific programming competences *and* to overcome syntactical issues.

References

Alphonce, C., & Ventura, P. (2002). Object orientation in CS1-CS2 by design. In *ITiCSE '02: Proceedings of the 7th annual conference on innovation and technology in computer science education*, Aarhus, Denmark (pp. 70–74). New York, NY: ACM Press.

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

Bahrick, H.P. (1984). Semantic memory content in permastore: Fifty years of memory for Spanish learned in school. *Journal of Experimental Psychology: General*, *113*, 1–47.

Bahrick, H.P., Bahrick, P., & Wittlinger, R.P. (1975). Fifty years of memories for names and faces: A cross-sectional approach. *Journal of Experimental Psychology: General*, *104*, 54–75.

Barnes, D.J., & Kolling, M. (2006). *Objects first with Java: A practical introduction using BlueJ* (3rd ed.). Essex, UK: Pearson.

Begel, A., & Simon, B. (2008). Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE technical symposium on computer science education, SIGCSE '08*, Portland, OR, USA (pp. 226–230). New York, NY: ACM.

Bennedsen, J. (2008). *Teaching and learning introductory programming – a model-based approach*. Norway: Department of Computer Science, University of Oslo.

Bennedsen, J., & Caspersen, M. (2004). Teaching object-oriented programming – towards teaching a systematic programming process. In *Eighth workshop on pedagogies and tools for the teaching and learning of object oriented concepts*, Oslo, Norway, June 14, 2004. Affiliated with 18th European Conference on Object-Oriented Programming, ECOOP 2004, Oslo, Norway.

Bennedsen, J., & Caspersen, M.E. (2007). Assessing process and product – A practical lab exam for an introductory programming course. *ITALICS, Innovation in Teaching and Learning in Information and Computer Sciences*, *6*, 183–202.

Bennedsen, J., & Caspersen, M. (2008). Model-driven programming. In J. Bennedsen, M. Caspersen, & M. Kölling (Eds.), *Lecture notes in computer science: Vol. 4821. Reflections on the teaching of programming: Methods and implementations* (pp. 116–129). Berlin: Springer-Verlag.

Bennedsen, J., Caspersen, M.E., & Kölling, M. (Eds.). (2008). *Lecture notes in computer science: Vol. 4821. Reflections on the teaching of programming, methods and implementations*. Berlin: Springer-Verlag.

Bjork, R. (1988). Retrieval practice and the maintenance of knowledge. In M. Gruneberg, P. Morris, & R. Sykes (Eds.), *Practical aspects of memory: Current research and issues* (Vol. 1, pp. 396–401). Chichester: Academic Press.

Butler, M., & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In R. Atkinson, C. McBeath, S.K.A. Soong, & C. Cheers (Eds.), *ICT: Providing choices for learners and learning. Proceedings Ascilite Singapore 2007* (pp. 99–107). Retrieved from http://www.ascilite.org.au/conferences/singapore07/procs/butler.pdf

Carbone, A., & Kaasbll, J.J. (1998). A survey of methods used to evaluate computer science teaching. In *ITiCSE '98: Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education*, Dublin City University, Ireland (pp. 41–45). New York, NY: ACM.

Caspersen, M.E. (2007). *Educating novices in the skills of programming*. Aarhus University, Department of Computer Science. Retrieved March 2012, from http://www.daimi.au.dk/~mec/dissertation/.

Caspersen, M.E., & Bennedsen, J. (2007). Instructional design of a programming course: A learning theoretic approach. In *ICER '07: Proceedings of the third international workshop on computing education research*, Atlanta, Georgia, USA (pp. 111–122). New York, NY: ACM.

Caspersen, M.E., & Kolling, M. (2009). STREAM: A first programming process. *Transactions on Computing Education*, *9*(1), 4:1–4:29.

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

Cooper, H., Valentine, J.C., Charlton, K., & Melson, A. (2003). The effects of modified school calendars on student achievement and on school and community attitudes. *Review of Educational Research*, *73*(1), 1–52.

Dickmann, R.A., & Lockwood, J. (1966). Computer personnel research group, 1966 survey of test use in computer personnel selection. In *Proceedings of the fourth SIGCPR conference on computer personnel research, SIGCPR '66*, Los Angeles, California (pp. 15–25). New York, NY: ACM.

Ebbinghaus, H. (1885a). *Über das Gedüchtnis*. New York, NY: Teachers College, Columbia University.

Ebbinghaus, H. (1885b). Memory: A contribution to experimental psychology. Retrieved May 14, 2009, from http://psychclassics.yorku.ca/Ebbinghaus/index.htm. Translated from German by Henry A. Ruger and Clara E. Bussenius (1913).
Engel, G., & Roberts, E. (2001). Computing curricula 2001 computer science, final report. Retrieved May 2012, from http://www.acm.org/education/curric_vols/cc2001.pdf. Technical report. East Lansing, MI: The Joint Task Force on Computing Curricula.
Fincher, S., & Petre, M. (2004). *Computer science education research*. London: Routledge Falmer.
Gamma, E., Helm, R., Johnson, R.E., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.
Gardiner, J.M., & Java, R.I. (1991). Forgetting in recognition memory with and without recollective experience. *Memory & Cognition*, *6*, 617–623.
Gorgone, J.T., Davis, G.B., Valacich, J.S., Topi, H., Feinstein, D.L., & Longenecker, H.E, Jr. (2002). IS 2002 – Model curriculum and guidelines for undergraduate degree programs in information systems. Technical report. Retrieved May 2012, from http://www.acm.org/education/education/curric_vols/is2002.pdf
Gorgone, J.T., Gray, P., Stohr, E.A., Valacich, J.S., & Wigand, R.T. (2006). MSIS 2006: Model curriculum and guidelines for graduate degree programs in information systems. *SIGCSE Bulletin*, *38*, 121–196.
Guzdial, M., & Forte, A. (2005). Design process for a non-majors computing course. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on computer science education*, St. Louis, Missouri (pp. 361–365). New York, NY: ACM.
Hilburn, T.B. (1993). 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.
Holmboe, C. (2005). *Language, and the learning of data modelling*. University of Oslo, Department of Teacher Education and School Development. Retrieved from http://urn.nb.no/URN:NBN:no-11609
Howe, E., Thornton, M., & Weide, B.W. (2004). Components-first approaches to CS1/CS2: Principles and practice. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, Norfolk, Virginia (pp. 291–295). New York, NY: ACM Press.
Jain, J., Cross, J.H, I., & Hendrix, D. (2005). Qualitative comparison of systems facilitating data structure visualization. In *ACM-SE 43: Proceedings of the forty-third annual Southeast regional conference* (pp. 309–314). Kennesaw, GA: ACM Press.
Kim, J., & Lerch, F.J. (1997). Why is programming (sometimes) so difficult? Programming as scientific discovery in multiple problem spaces. *Information Systems Research*, *8*(1), 22–50.
Knudsen, J.L., & Madsen, O.L. (1988). Teaching object-oriented programming is more than teaching object-oriented programming languages. In S. Gjessing, & K. Nygaard (Eds.), *ECOOP '88 European Conference on Object-Oriented Programming*, Oslo, Norway, August 15–17, 1988 (pp. 21–40). Berlin: Springer-Verlag.
Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Computer Science Education*, *13*, 249–268.
Kumar, A.N. (2004). Using online tutors for learning what do students think? In *Proceedings of the 34th ASEE/IEEE frontiers in education conference*, October 20–23, 2004 (pp. T3F–9 – T3F–13). Retrieved from http://fie-conference.org/fie2004/papers/1217.pdf.
Levy, R.B.B., Ben-Ari, M., & Uronen, P.A. (2003). The Jeliot 2000 program animation system. *Computers & Education*, *40*(1), 1–15.
Loftus, E. (1980). *Memory: Surprising new insights into how we remember and why we forget*. Reading, MA: Addison-Wesley.
Macken, W.J., & Hampson, P. (1991). Integration, elaboration, and recollective experience. *The Irish Journal of Psychology*, *14*, 270–285.
Malmi, L., Karavirta, V., Korhonen, A., Nikander, J., Seppl, O., & Silvasti, P. (2004). Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, *3*, 267–288.

Matzko, S., & Davis, T.A. (2006). Teaching CS1 with graphics and C. In *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on innovation and technology in computer science education*, Bologna, Italy (pp. 168–172). New York, NY: ACM Press.

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming views of students and tutors. *Education and Information Technologies*, *7*, 55–66.

Montgomery, D.C., & Peck, E.A. (1982). *Introduction to linear regression analysis*. New York, NY: John Wiley.

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bulletin*, *39*, 151–155.

Newell, A., Rosenbloom, P.S., & Anderson, J.R. (1981). Mechanisms of skill acquisition and the law of practice. In J.R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 1–55). Hillsdale, NJ: Erlbaum.

Newell, A., Rosenbloom, P.S., & Laird, J.E. (1989). Symbolic architectures for cognition. In Michael I. Posner (Ed.), *Foundations of cognitive science* (pp. 93–131). Cambridge, MA: MIT Press.

Pattis, R.E. (1993). The procedures early approach in CS 1: A heresy. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on computer science education*, Indianapolis, Indiana (pp. 122–126). New York, NY: ACM Press.

Reek, M.M. (1995). A top-down approach to teaching programming. In *SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on computer science education*, Nashville, Tennessee (pp. 6–9). New York, NY: ACM Press.

Reges, S. (2000). Conservatively radical Java in CS1. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on computer science education*, Austin, Texas (pp. 85–89). New York, NY: ACM Press.

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

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

Schmolitzky, A. (2004). ''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*, Vancouver, British Columbia (pp. 64–67). New York, NY: ACM Press.

Schulte, C., & Bennedsen, J. (2006). What do teachers teach in introductory programming?. In *Proceedings of the second international workshop on computing education research, ICER '06*, Canterbury (pp. 17–28). New York, NY: ACM.

Sedgewick, R., & Schidlowsky, M. (1998). *Algorithms in Java, Parts 1–4: Fundamentals, data structures, sorting, searching* (3rd ed.). Boston, MA: Addison-Wesley Longman Publishing Co.

Shackelford, R., Cross II, J.H., Davies, G., Impagliazzo, J., Kamali, R., LeBlanc, R., ... Topi, H. (2006). The Overview Report. Retrieved June 2009, from http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf

Simon, B., Lister, R., & Fincher, S. (2006). Multi-institutional computer science educational research: A review of recent studies of novice understanding. In *Proceedings of the 36th Annual Frontiers in Education Conference*, October (pp. SE412–17). Retrieved from http://fie-conference.org/fie2006/papers/1149.pdf

Simpson, D. (1973). Psychological testing in computing staff selection: A bibliography. *ACM SIGCPR Computer Personnel*, *4*(1–2), 2–5.

Slamecka, N.J., & McElree, B. (1983). Normal forgetting of verbal lists as a function of their degree of learning. *Journal of Experimental Psychology: Learning, Memory & Cognition*, *9*, 384–397.

Soldan, D., Hughes, J.L., Impagliazzo, J., McGettrick, A., Nelson, V.P., Srimani, P.K., & Theys, M.D. (2004). *Computer engineering 2004 – Curriculum guidelines for undergraduate degree programs in computer engineering*. Retrieved from http://www.acm.org/education/education/curric_vols/CE-Final-Report.pdf

Squire, L. (1999). On the course of forgetting in very long-term memory. *Journal of Experimental Psychology: Learning, Memory & Cognition*, *15*, 241–245.

Stasko, J., Badre, A., & Lewis, C. (1993). 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* (pp. 61–66). Amsterdam: ACM.

Stein, L.A. (1998). What we swept under the rug: Radically rethinking CS1. *Computer Science Education*, *8*, 118–129.

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

Tulving, E. (1985). Memory and consciousness. *Canadian Psychologist*, *25*, 1–12.

Weinberg, G.M. (1971). *The psychology of computer programming*. New York, NY: Van Nostrand Reinhold.

Weinberg, G.M. (1998). *The psychology of computer programming*. New York, NY: Dorset House Publishing.

Wickelgren, W. (1972). Trace resistance and the decay of long-term memory. *Journal of Mathematical Psychology*, *9*, 418–455.

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

Wolz, U., & Koffman, E. (2000). 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*, Ramapo College of New Jersey, Mahwah, New Jersey (pp. 1–3). USA: Consortium for Computing Sciences in Colleges.

Woodworth, R. (1938). *Experimental psychology*. New York, NY: Henry Holt.

Appendix A: Pre-test

Aarhus University
Dept of computer science

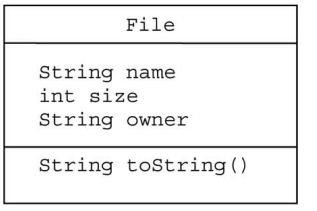
APPENDIX

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"
   ```

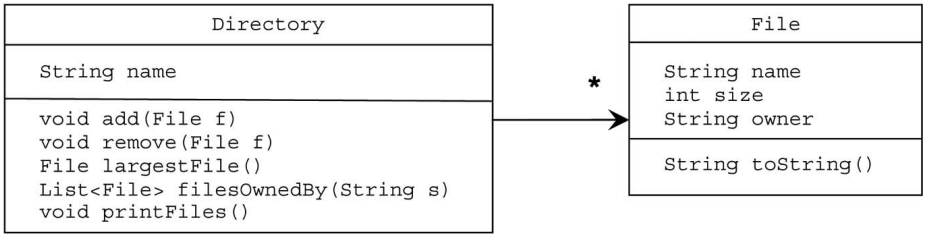


2. Create a *Driver*-class containing an *exam*-method. The method must be static, have return type void and no parameters.

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.

Appendix B: Post-test

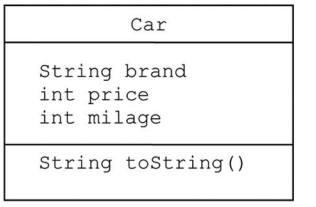
Aarhus Universitet
Datalogisk Institut

Eksperiment i dIntProg
3 juni 2009

Experiment 2, (post-test)

1. Create a class, *Car*, representing a car; the class *Car* 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 car, e.g..

   ```
   "Volvo V50, 287000 kr, 57000 km"
   ```

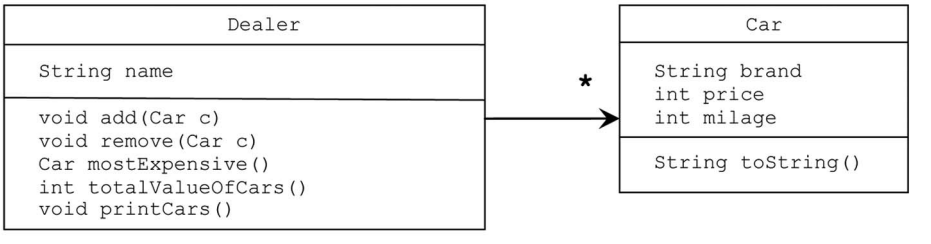


2. Create a Driver-class containing an exam-method. The method must be static, have return type void and no parameters.

3. Create three *Car*-objects, using object-references *c*1, *c*2 og *c*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, *Dealer*, representing a used-car dealer; the class *Dealer*, and its rtelation to the class *Car*, is specified in the following UML-diagram:



5. Program the methods *add* and *remove* who respectively adds and removes the *Car*-objekt *c* to/from the *Dealer*-object.

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

7. Program the method *mostExpensive*. The method must return the most expensive car (it can be assumed that the dealer has at least one car for sale; if several cars are equally expensive it is subordinate which car that is returned). Extend the *Car*-class with the necessary get-methods.

8. Use the method *mostExpensive* in the *exam*-method in the *Driver*-class, to print out information about the most expensive car at the car-dealer.

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

9. Program the method *totalValueOfCars*. The method must return the total amount that all the cars at the given dealer costs.er. Extend the *Car*-class with the necessary get-methods.

10. Program the method *printCars*. The method must print out a list of all the cars at a dealer ordered by price.

Call the observer and demonstrate your final solution.