# Assessing Process and Product –A Practical Lab Exam for an Introductory Programming Course[1]

Jens Bennedsen
IT University West
Fuglsangs Allé 20
DK- 8210 Aarhus V, Denmark
jbb@it-vest.dk

Michael E. Caspersen
Department of Computer Science
University of Aarhus
DK-8200 Aarhus N, Denmark
mec@daimi.au.dk

**Abstract -** The final assessment of a course must reflect its goals and contents. An important goal of our introductory programming course is that the students learn a systematic approach for the development of computer programs. Having the programming process a as learning objective naturally raises the question how to include this in assessments. Traditional assessments (e.g. oral, written, or multiple choice) are unsuitable to test the programming process.

We describe and evaluate a practical lab examination that assesses the students' programming process as well as the developed programs. The evaluation is performed in three ways: by analyzing the results of four lab examinations (with more than 1100 students), by semi-structured individual interviews with representatives of the involved persons (students, teaching assistants (TAs), lecturer, and examiner) and by an experiment to evaluate the impact of the assignment text on the programming process.

The result of the evaluation is encouraging and indicates the value of alignment and strong conformity between goal, content and assessment of the introductory programming course.

Keywords: introductory programming, teaching programming, practical lab examination.

[1] This paper is an expanded and revised version of a paper presented at Frontiers in Education Conference in 2006 (Bennedsen & Caspersen, 2006).

## 1 Introduction

There are many alternative assessment methods. Stiggins (2005) enumerates four main categories: *selected response* (multiple choice questions and short answer questions), *essay* (poster presentation, written report), *performance assessment* (case study, practicum, project, and reflective journal/diary), and *personal communication* (class presentation, interview, and learning contract). A typical oral examination is classified as personal communication (interview), and a written examination is classified as selected response (short answer question). According to Stiggins (2005), each category has advantages in assessing different learning outcomes. For example, a selected response assessment task, such as a series of multiple-choice questions, is able to assess all areas of mastery of knowledge but only some kinds of reasoning.

Multiple choice tests have recently been coming into favour as a useful evaluation tool at the university level (Brown, 2001; Roberts, 2006; Woodford & Bancroft, 2005), in contrast to the earlier view that they support only superficial learning (Biggs, 2003). A recent result regarding multiple choice is that five self-evident axioms are sufficient to determine completely the unique correct scoring strategy for multiple choice tests where students are allowed to check several boxes to convey partial knowledge (Frandsen & Schwartzbach, 2006).

Based upon Bloom's classification of educational objectives (Bloom, Krathwohl, & Masia, 1956), Lister and Leaney have developed a criterion-referenced grading scheme to cope with diversity among students in an introductory programming class. In the traditional norm-referencing approach to grading, all students attempt the same programming tasks, and the attempts are graded "to a curve". The danger is that such tasks are aimed at a hypothetical average student. Weaker students can do few of these tasks and learn little. Meanwhile, these tasks do not stretch the stronger students, so they too are denied an opportunity to learn: *Our contribution has been to bring disparate grading techniques together, uniting them in a coherent grading philosophy* (Lister & Leaney, 2003).

Plagiarism is a major issue in computer science education as exposed by an ITiCSE 2002 working group (Dick et al., 2002). Several assessment systems incorporate plagiarism detection; Lancaster & Culwin (2004) provide a comparison of source code plagiarism detection engines. Daly and Horgan (2005) present a system based on watermarks, allowing them to distinguish supplier and recipient .

Some research has been aimed at understanding whether the assessment is valid, i.e. whether it represents the kind of knowledge the educator wants it to assess (Fincher & Petre, 2004). An example of this concerns assessment of early programming competence. With reference to the findings of McCracken et al. (2001), which shows that many computing students are not able to develop straightforward programs after the introductory programming sequence, Daly & Waldron (2004) argue that normal student assessment should have highlighted this problem; since it did not, normal assessment of programming ability does not work. The authors examine why current assessment methods (written exams and programming assignments) are faulty and investigate another method of assessment: the lab exam. Furthermore, the authors show that this form of assessment is more accurate, and they explain why accurate assessment is essential in order to encourage students to develop programming ability.

The final assessment must reflect aims, goals, and contents of a course (Biggs, 2003; Prior & Lister, 2004). John Biggs developed constructive alignment – based on the work of Ralph Tyler (1949) – to help teachers focus on their development of courses where the learning outcome of students was higher. In constructive alignment, according to Biggs (2003), we first state the learning outcomes we intend our students to achieve. The outcome statements contain a learning activity, a *verb* that students need to perform to properly achieve the outcome. That verb (e.g. 'explain', 'apply', 'reflect') then needs to be activated by the teaching and learning activities we give students: lecturing to them usually does not do that. The assessment tasks should also require students to enact that same verb. How well they solve those problems is the authentic assessment, not sitting exams about what we have taught. That verb is what achieves alignment: it is in our intended outcomes, in the teaching and learning activities, and in the assessment tasks. Traditionally, educational systems are not aligned. The curriculum is usually a list of topics telling teachers what to 'cover', the default teaching method is the lecture, in which students are told about the topic — they do not have to *enact* their understanding

An important goal of our introductory programming course is that the students learn a systematic approach to the development of computer programs. Learning a systematic approach to programming implies that the students must gain a clear understanding of the programming process and the activities that are part of this process. They must also develop the ability to apply these to develop programs.

Recognizing the importance of programming techniques and the programming process when designing a programming course implies the need for adoption of a suitable assessment form. Traditional assessment forms (e.g. oral or written examinations, multiple choice questions) are unsuitable to test the programming process.

Another equally important argument for assessing the programming process is that *the spirit and style of student assessment defines de facto the curriculum* (Rowntree, 1977 p.1). Ramsden makes a similar observation *the type of grading influences the student's learning approach* (Ramsden, 1992).

The bottom line is that it is essential to apply an evaluation form where the students demonstrate their practical programming skills as well as their understanding of the fundamental concepts and theories from the curriculum of the course. Consequently, we need to develop a new type of assessment suitable to test the programming process as well as the product.

The characteristics of the lab examination described and evaluated in this paper are that it

i. provides a valid and accurate evaluation of the student's programming capabilities,
ii. evaluates the process as well as the product,
iii. encourages the students to practice programming throughout the course,
iv. can be used to assess 120-140 students per. day, and
v. plagiarism is impossible.

The rest of the paper is structured as follows: Section 2 describes the context of the lab examination. Section 3 gives a more thorough description of the final lab examination. Section 4 presents and discusses the findings from the evaluation of the lab examination. In section 5 we discuss related and future work. The conclusions are drawn in section 6.

## 2 Goals, Content and Assessment

To provide an understanding of the context, this section describes goal, form, and content of the introductory programming course.

### *2.1 General Information*

Our programming course spans the first half of CS1 at University of Aarhus. The course runs for seven weeks, and after the course there is a 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 during the course is a prerequisite for the final exam but does not count as part of the grading.

There are approximately 300 students per year from a variety of study programmes, e.g. computer science, mathematics, geology, nano science, economy, multimedia. 40% of the students are majors in computer science, and they are the only group of students that continue with the second half of CS1. The rest of the students proceed to other programming courses related to their fields (e.g. multimedia programming, scientific computing) if they proceed with programming at all.

The students are grouped in teams of 18-20 students; typically there are 13-14 teams per year. Each team has its own teaching assistant (TA) – a PhD or MSc student.

### *2.2 Goals*

The purpose of the course is that students learn the foundation of systematic construction of simple programs and through this obtain knowledge about the role of conceptual modelling in object-oriented programming. Furthermore, it is the goal that students become familiar with a modern programming language, fundamental programming language concepts, and selected class libraries.

After the course the students must be able to explain and use fundamental elements in a modern programming language, use conceptual modelling in relation to preparing simple object-oriented programs, implement simple object-oriented models in a modern programming language, and use selected class libraries. A description can be found at http://mit.au.dk/da/studier/udbudbeskriv.cfm?udbudid=6047&lan=en&scope=1&parentelem=8647

### *2.3 Form*

The course runs for seven weeks. Every week there are four lecture hours and four lab hours with a TA. In addition to the scheduled hours, students work approximately seven hours per week in study groups or on their own.

The four lecture hours per week are used for presentation and discussion of general concepts and the programming process. The programming process is revealed through live programming in front of the students in the lecture theatre using computer and projector and through process recordings (narrated, screen-captured video recordings of program development sessions); see (Bennedsen & Caspersen, 2005).

Every week (except for the first) there is a mandatory assignment that must be submitted to the TA. The TA examines the assignments and gives personal as well as collective feedback to the students. Approval of five out of six weekly assignments is a prerequisite for the final exam but does not count as part of the grading. The weekly assignments are primarily used to keep the students up to the mark on the practice of programming.

### *2.4 Content*

The course content is fundamental programming language concepts, object-orientation, and techniques for systematic construction of simple programs.

- **Fundamental programming language concepts**: variable, value, type, expression, object, class, encapsulation, control structure, method/procedure, recursion, type hierarchies.

- **Object-orientation**: modelling; class structures (specialization, aggregation and association); use of selected class libraries (in particular collection libraries), interfaces and abstract classes.
- **Systematic development of small programs**: modularization, stepwise refinement/incremental development, test.

This is a logical listing of the course contents; it is *not* the order in which the content is covered. The content is covered using a spiral approach (Bergin, 2006), for further details of the structure and content of the course see (Bennedsen & Caspersen, 2004; Caspersen & Christensen, 2000). The programming process we teach our students is called STREAM (*stubs*, *tests*, *representation*, *evaluation*, *attributes*, and *methods*) (Caspersen & Kölling, 2006). This is a test driven, incremental process with five steps:

1. Create the class (with method stubs)
2. Create tests
3. Alternative representations
4. Instance fields
5. Method implementation

We teach the students general coding recipes to more or less automatically handle step 1, how to create a representation evaluation matrix focusing on the estimated effort it takes to implement each method using a particular object representation and algorithmic patterns (Muller, 2005) to solve the implementation of the methods.

## 3 Assessment through a Lab Examination

This section discusses the examination requirements, the organization of the lab examination and the actual lab examination.

### *3.1 Conformity between Goals, Content, and Assessment*

As mentioned in section 2.2 the goals of the course are that the student must be able to explain and

- *use* fundamental elements in a modern programming language,
- *use* conceptual modelling in relation to preparing simple object-oriented programs,
- *implement* simple object-oriented models in a modern programming language, and
- *use* selected class libraries.

During the course, as in real life, programs are developed using a standard development environment running on a computer. An ordinary written exam with pen and paper is an artificial situation and therefore insufficient and inappropriate to test the student's ability to develop programs. For the same reasons an ordinary oral examination and a multiple choice test would be inappropriate.

To ensure alignment and maximum conformity between goals, content, and assessment we have designed a practical examination organized in a lab.

### *3.2 Organization of the Lab Examination*

The examination resembles an ordinary lab session. 25 students are tested concurrently. We schedule one hour per group of 25 students, but only 30 minutes for the actual lab examination. The rest of the time is used for administrative activities and as buffer.

Each group of students receives a different assignment consisting of nine small progressive programming tasks. In principle the assignments are identical (they are all instances of the same generic assignment), but the students does not know nor realize this. The similarity of the assignments is important for fairness as well as comparability of the students' results. The sample assignment in Figure 1 deals with sales notes and sales persons; other exercises concern luggage and flights, employees and departments, museums and paintings, etc. Although the concepts modelled by the classes vary, the assignments have similar structure.

---

**Exam (30 min. lab exam)**

1. Create a class *SalesNote* representing a sales note; the class *SalesNote* is specified in the UML diagram on the right. The three field variables must be initialized in a constructor (via parameters of suitable types). The method *toString* must return a string representation of a sales note, e.g.

```
+----------------------+
| SalesNote            |
+----------------------+
| String date          |
| String description   |
| int amount           |
+----------------------+
| String toString()    |
+----------------------+
```

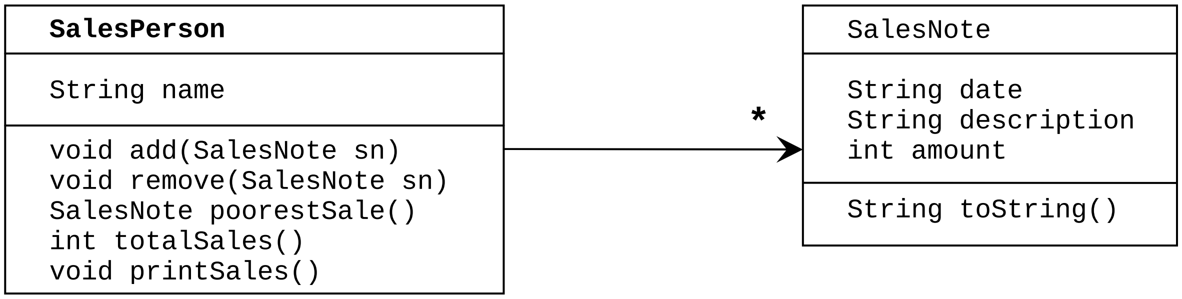
   "2005-10-24, A4 paper, 237 kr."

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

3. Create two *SalesNote* objects, via object references *sn1* and *sn2*, in the *exam* method and print these using the *toString* method.

**Call one of the supervisors for review of what you have created so far; prepare the presentation by opening the relevant files and place them next to each other on the desktop.**

4. Create a new class, *SalesPerson*, representing a sales person; the class *SalesPerson* and its relation to the class *SalesNote* is specified in the following UML diagram:



5. Program the methods *add* and *remove* to respectively add/ remove the *SalesNote* object *sn* to/from the *SalesPerson* object.

6. Create an object of type *SalesPerson* in the *exam* method in the *Driver* class and associate the two already created *SalesNote* objects to the *SalesPerson* object.

7. Program the method *poorestSale*. The method must return the *SalesNote* object representing the poorest sale (it can be assumed that a sales person has had at least one sale). If more *SalesNote* objects have the same minimum value, it does not matter which is returned. Add the necessary get methods to the *SalesNote* class.

8. Use the method *poorestSale* (*SalesPerson*) and *toString* (*SalesNote*) for printing the sales note representing the sale of poorest value (in the *exam* method in the *Driver* class).

**Call one of the supervisors for review of what you have created so far; prepare the presentation by opening the relevant files and place them next to each other on the desktop.**

9. Program the method *totalSales* (*SalesPerson*). The method must return the sum of all sales for the sales person concerned. Test the method from the *exam* method in the *Driver* class.

10. Program the method *printSales*. The method must print a list of all sales of a sales person. The list must be chronologically ordered (sales from the same day must be ordered according to the value of the sales). Hint: Let the class *SalesNote* implement the *Comparable* interface.

**Figure 1: Prototypical assignment**

### *3.3 Assessment of Product and Process*

In the test for completed apprenticeship of traditional crafts, the examiner inspects the apprentice while they construct their exam product; the quality of the apprentice's construction process as well as the quality of the final product counts in the final grading.

Because of similar goals regarding the assessment of process and product, we have adopted a similar examination form where the lecturer and the external examiner evaluate the programming process as well as the program produced by each student by inspecting the students during the examination.

The students' behaviour as well as the quality of the programs they produce count in the final grading but not on equal footing. An appropriate and systematic programming process can compensate for minor flaws and errors in the product and result in a pass mark for the student (as described above we have a binary grading scheme), and similarly a poor process can be the determining factor when the product is on the edge. Although we emphasize the programming process, it is not the case that a nice product will be turned down due to a poor process (which is unlikely anyway). In order to demonstrate that a student fulfils the goals of the course, the student must implement at least the association (in Figure 1 this is programming task five and six) and a method that iterates through the objects (programming task seven and eight). The minor flaws mentioned above can be minor errors in e.g. the iteration (the student finds the maximum in stead of the minimum) or a simple syntax error but the overall implementation is acceptable.

To avoid practical problems during start-up and finalization of the lab examination (e.g. login problems, applying naming conventions, delivery of the exam products), and to ensure that minor unimportant programming errors, tool problems, etc. do not hinder the student's problem solving and programming, five TAs are present during the lab examination to support the students. If the TAs have doubts about their role (e.g. how much to interact with the students), they consult the lecturer or external examiner on-the-fly.

To let the students settle down and get started, they are not inspected until they have passed a checkpoint after the first three programming tasks. The students are instructed to call upon a TA or the lecturer when they reach the checkpoint to show and demonstrate their solution. The time when the first three tasks are done is noted. When a student has passed the checkpoint, the lecturer and external examiner start inspecting the student's behaviour. The poorest students never reach the checkpoint and therefore, the inspection time is focused on those students who have a chance of passing. To allow for efficient inspection, the students are instructed to keep all editor windows open and tiled on the screen.

From several years of experience, we know that most of the students five to seven minutes after the first checkpoint are implementing the iteration. Around that time the student is monitored more closely, and after a short inspection of the students programming process, it is usually possible to determine the pass grade. This is a very efficient way to know when and in what order to look at the students' solutions. This is also a method to ensure that the students have some silence and can concentrate during the exam.

The elements we look for are among other things:

- **Errors**: How does the student handle an error? Does the student pay attention to the error message? Does the student address the error in a systematic way? Does the student use the debugger in a reasonable way?
- **Documentation**: Does the student use the documentation when in doubt?
- **Programming patterns**: Does the student use standard patterns to solve the problems (e.g. a findOne associated object (Caspersen & Bennedsen, 2007) pattern to solve programming task 7 above)
- **Conceptual model**: Does the program structure reflect the UML class model?
- **Handling of tools**: Does the student handle the programming environment in a reasonable manner?
- **Testing**: Does the student test the program?
- **Specification**: Does the student take the specification into account? (e.g. in programming task 7 the prerequisite is that there is at least one SalesNote object)

## 4 Evaluation

In this section, we present and discuss an evaluation of the lab examination described above.

### *4.1 Evaluation Method*

The evaluation of the lab exam was performed in two ways: by analyzing the results of four consecutive lab examinations (2003, 2004, 2005 and 2006) and by semi-structured individual interviews with students, TAs, the examiner, and the lecturer.

### *4.2 Quantitative Evaluation*

For each of the three years we have collected data about the students for four variables (and two derived). The description of the variables can be found in Table 1.

| **Variable** | **Description** |
|---|---|
| *students* | students enrolled for the course |
| *abort* | students that aborted the course before the final exam |
| *exam* | students allowed to take the final exam |
| *skip* | students that did not show up for the final exam but was allowed to |
| *fail* | students who failed the final exam |
| *pass* | students who passed the final exam |

**Table 1**: *Description of type of data*

The numbers in table 1 are related as follows:

*students* = *abort* + *exam*

*exam* = *skip* + *fail* + *pass*

From these numbers we calculate *exam rate*, *pass rate* and *retention rate* (*exam*/*students*, *pass*/*exam*, *pass*/*students*). The results are presented in Table 2.

| | **2003** | **2004** | **2005** | **2006** |
|---|---|---|---|---|
| *students* | 276 | 220 | 295 | 326 |
| *abort* | 63 | 26 | 28 | 44 |
| *exam* | 213 | 194 | 267 | 282 |
| *exam rate* | 77.2 % | 88.2 % | 90.5 % | 86.5 % |
| *skip* | 13 | 5 | 3 | 1 |
| *fail* | 15 | 19 | 29 | 17 |
| *pass* | 185 | 170 | 235 | 264 |
| *pass rate* | 86.9 % | 87.6 % | 88.0 % | 93.6 % |
| *retention rate* | 67.0 % | 77.3 % | 79.7 % | 81.0 % |

**Table 2**: *Statistics from three years of practical lab exams*

The figures in Table 2 reveal two interesting aspects: the improved exam rate (and retention rate) from 2003 to the following years, and the high pass rate in general.

The curriculum was radically redesigned in 2003 going from a semester structure to a quarter structure; consequently the traditional CS1 course was split in two courses with an exam in between. The students of 2003 were the first to take the new course with the new examination form, and therefore there were no traditions for the students to lean on. In the following years (2004-2006) the students have had the old exam questions to use for practice, and older students to hear war stories from. In the following years the lecturer could be more explicit when describing the requirements for the exam and the exam form. We believe that this is the primary reason for the improved exam rate.

The pass rate is high compared to what others report (Andersson & Roxå , 2000; Börstler, Johansson, & Nordstrom, 2002). We believe that this primarily is due to the alignment and the strong conformity between goal, content and assessment of the course.

## *4.3 Qualitative Evaluation*

The semi-structured interviews were conducted two to three weeks after the final exam. Ten students were selected to get a mixture of major and gender. One interviewer conducted each interview. The interviews were audio taped for later analysis. The interviews followed an interview guide focusing on three topics: the lab exam form in general, this specific exam, and the evaluation form compared to other evaluation forms. In the analysis that follows, quotations from the interviews are presented that describes the general attitude of the group. The interviews were done in Danish, and the quotations translated into English by the authors.

### *4.3.1 The Students*

There was a very little difference in the way that the interviewed students had experienced the lab exam; their answers were largely similar. We find therefore that the students are representative of the general attitude towards the exam, although we cannot be sure.

All of the interviewed students found the evaluation form fair. They defined *fair* as *if you have practiced during the course, you can expect to pass the exam*. They all found that the form and content of the exercise was very adequate with respect to the goals of the course. As one student noticed: *Programming requires very abstract thinking, but it is also a craft ... the examination form perfectly suits this mixture*

One of the students did not like it that a TA was looking over her shoulder. She felt insecure and nervous. However, she was the only one having this experience – no one else minded having the TAs around (some even found their presence to give more peace of mind).

The examination incited the students to practice programming. As an option for the students, exam exercises from the previous year were available for preparation for the exam. Almost all of the students interviewed indicated that they had solved most of the previous exam exercises when they prepared for the exam; as one student replied when asked about his preparations: *I solved all the [old] exam exercises*.

Students were instructed to call the TA after solving the first three tasks of the exercise (Figure 1) to demonstrate what they had achieved. None of the students found this to be problematic, but some of them pointed to the possible problem, that the slow students might feel this as an extra stress factor (knowing that many of the other students have finished). In conclusion, only one of the interviewed students felt the examination to be stressful.

All of the interviewed students felt that a more fine-grained marking could take place, but it would require more time and more tasks. Most thought that one hour would be sufficient for this.

#### *4.3.2 The Teaching Assistants*

The interviews with the teaching assistants in many ways supported the statements from the students. They also found the exam to be fair and had the impression that it evaluates the students programming skills.

In the beginning, the TAs had some difficulties knowing to what extent they could answer questions. During the exam, the TAs developed a practice: they helped a student who had spent several minutes trying to figure out a simple problem, but did not help with problems that were more fundamental. If in doubt, the TAs asked the lecturer or examiner. Apart from this, they did not feel uncomfortable with they role.

#### *4.3.3 The Lecturer and the External Examiner*

Both the lecturer and the examiner found the exam form to be both fair and evaluating the learning objectives of the course. The external examiner found that the exam evaluated the student's understanding of the general concepts although it was impossible to evaluate that the student was *able to explain [...] fundamental elements in a modern programming language*. They found that it was easy to assess an objective pass/fail criterion due to the generic exercises. The examiner thought that a little longer time would give an even better evaluation criterion.

The examination gave a good impression of the students programming skills including their programming process. As the examiner said: *When you get an error message from the compiler you must be able to figure out what is wrong … that is a part of a practical programming skill*.

### *4.4 Evaluation of the Evaluation of the Process*

In section 3.2, we demonstrated how careful and detailed phrasing of an assignment can guide the students through an incremental programming process characterised by a (more or less) monotone development trace.

We have used similar phrasing for the final exam in order to ease evaluation of the students (the more control we have of their process, the easier it is to evaluate progress). This raises the question of how well we really can evaluate the students programming process; if we provide detailed guidance, how then can the students demonstrate their personal competence on this issue?

In order to evaluate the learning effect specifically with respect to process competence, we set up an experiment just prior to the previous final examination. We designed a programming task where no guidance at all was provided; the assignment consisted of a class model and functional specifications of the methods of the classes. Apart from the lack of a detailed description, the content and level of the assignment were comparable with the general assignments as illustrated in Figure 1. The assessment used for the experiment can be found in the Appendix.

All students participating in the course in fall 2006 were invited to take part in the experiment, and 38 students accepted the invitation (the students who accepted the invitation were representative of the whole population with respect to major).

Our goal was to evaluate the students programming process now that no guidance was provided. A group of TAs examined the students and took notes of their behaviour; the student/TA ratio was 3/1. The TA's were instructed to make notes of the students' programming process. In particular, they should make a note whenever a student violated the 'standard process' that had been taught in the course (demonstrated through live programming and several videos of worked examples).

The conclusion of the experiment was that all students followed the process they had been taught even though no guidance was provided. They developed one part of the program at a time nicely separating the different concerns of the task. There was some variation as to how frequent the students swapped between writing test code and writing production code and as to whether they wrote the test code before or after the production code. The programming process we teach the students suggests writing test code before the production code, but almost all the students wrote the production code first.

Interestingly, hardly any of the students took the easy way out by implementing *Comparable* in order to get away with trivial implementations of three of the requested methods.

Immediately after the practice exam, we conducted informal interviews with groups of students. When asked about their testing behaviour (less frequent and after the fact), they responded that they did not feel the need for the test in order to implement the requested methods; they wrote the test because they had to, and not because the needed it to understand the task. It is hard to blame them on that because their behaviour mirrors expert behaviour (Robillard, 2005; Winslow, 1996).

We refrain from drawing too strong conclusions from this experiment, but the result suggests that students do learn the process we teach —at least when they are exposed to familiar tasks. But, again, this is just as it is with experts. Winslow (1996) puts it this way: *Experts, when given a task in* *a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general (opportunistic) problem solving* (p. 18).

### *4.5 Concluding the Evaluation*

The exam tests the process as well as the product. In some cases the process was the decisive factor. One special example of this was a student that was ill and therefore worked very slowly; however slow, her programming process was very good demonstrating a systematic approach to solving the problems.

The evaluation indicates that the lab examination supports the learning objective of the course. The students and the lecturer/examiner consider the lab examination fair. The assessment does not require many resources: 250 students can be handled using less than 90 person-hours.

Low retention is one of the main problems in CS1 courses. As noticed by Kumar (2003 p. 40) their retention "has been around 50%". In this course, the retention is above 75%. We have found that the examination form kept the students up to the mark; they did actually practice programming. We think this is one of the explanations of the relatively high retention rate.

For computer science students the examination form must be seen in conjunction with the examination form of the following course (the second part of CS1), in which an oral examination focusing more on the conceptual aspects of introductory programming is used. There is a progression from the first exam to the next, from testing practice to testing conceptual knowledge.

## 5 Related Work

Recently, a growing number of papers reporting on laboratory exams for introductory programming courses have been published (Barros, Estevens, Dias, Pais, & Soeiro, 2003; Califf & Goodwin, 2002; Chamillard & Braun, 2000; Daly & Horgan, 2005). All report good results using this apparently novel assessment form. However, a common characteristic of the assessment methods presented in these articles, and a deficiency compared to the method described herein, is that the evaluation and grading is based solely upon the end product, the students' final solutions.

In (Califf & Goodwin, 2002) the authors describe the grading in their lab final (their word for lab exam): *Grading on the exam is focused on working programs*. Only the result of the process is evaluated, not the process. Barros reports on the use of lab exams during the course, but the final exam is a traditional written exam. The *rationale behind maintaining code written in the final exam was to evaluate the students in an environment where trial and error is simply not possible* (2003, p.18). Again, they do not include an evaluation of the programming process in their lab exam; the focus is on the final product only.

## 6 Conclusion

We have described and evaluated a lab exam which has a number of advantages. It is simple to evaluate the student's programming process as well as the product (the result of the student's efforts). It is a fair and effective exam. We use standardized exercises that each covers more than 80% of the curriculum. The environment for the exam is the normal daily work environment. It is a lightweight exam easy to prepare and carry out. It requires a couple of days to prepare the exercises for the exam, and we had a throughput of 100 students per day. Everyone involved, in particular the students, regarded the form as well as the content of the exam to be very good and in excellent correspondence with the learning objectives of the course.

## 7 Acknowledgement

It is a pleasure to thank Gudmund Frandsen for valuable comments during development and practice of the lab exam described and evaluated in this paper. We will also like to thank the students and TAs who participated in the interviews. A special thank to Michael Kölling and the anonymous reviewers for valuable comments on earlier versions of this article.

## 8 References

Andersson, R., & Roxå , T. (2000). Encouraging students in large classes. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 176-179

Barros, J. P., Estevens, L., Dias, R., Pais, R., & Soeiro, E. (2003). Using lab exams to ensure programming practice in an introductory programming course. *ITiCSE '03: Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education,* Thessaloniki, Greece. 16-20.

Bennedsen, J., & Caspersen, M. E. (2004). Programming in context: A model-first approach to CS1. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, USA. 477-481.

Bennedsen, J., & Caspersen, M. E. (2005). Revealing the programming process. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* St. Louis, Missouri, USA. 186-190.

Bennedsen, J., & Caspersen, M. (2006). Assessing process and product — A practical lab exam for an introductory programming course. *Proceedings of the 36th Annual Frontiers in Education Conference,* San Diego, California. M4E-16-M4E-21.

Bergin, J. (2006). *Pedagogical patterns.* Retrieved January 21st, 2007, from http://csis.pace.edu/~bergin/#pedpat

Biggs, J. B. (2003). *Teaching for quality learning at university* (Second ed.) Berkshire, United Kingdom: Open University Press.

Bloom, B. S., Krathwohl, D. R., & Masia, B. B. (1956). *Taxonomy of educational objectives. the classification of educational goals. handbook I: Cognitive domain*. New York: Longmans, Green.

Börstler, J., Johansson, T., & Nordstrom, M. (2002). Teaching OO concepts-a case study using CRC-cards and BlueJ. *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference,* Boston, Mass. T2G-1-T2G-6.

Brown, R. W. (2001). Multi-choice versus descriptive examinations. Proceedings of 31st Annual Frontiers in Education Conference, 2001. Reno, NV, USA. T3A-13-T3A-18.

Califf, M. E., & Goodwin, M. (2002). Testing skills and knowledge: Introducing a laboratory exam in CS1. *SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education,* Cincinnati, Kentucky. 217-221.

Caspersen, M. E., & Bennedsen, J. (2007). Instructional design of a programming course - A learning theoretical approach. Proceedings of the Third International Computing Education Research Workshop. Atlanta, GA, USA. To appear.

Caspersen, M. E., & Christensen, H. B. (2000). Here, there and everywhere — on the recurring use of turtle graphics in CS1. *ACSE '00: Proceedings of the Australasian Conference on Computing Education,* Melbourne, Australia. 34-40.

Caspersen, M. E., & Kölling, M. (2006). A novice's process of object-oriented programming. *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications,* Portland, Oregon, USA. 892-900.

Chamillard, A. T., & Braun, K. A. (2000). Evaluating programming ability in an introductory computer science course. *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education,* Austin, Texas, United States. 212-216.

Daly, C., & Horgan, J. (2005). Patterns of plagiarism. *SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education,* St. Louis, Missouri, USA. 383-387.

Daly, C., & Waldron, J. (2004). Assessing the assessment of programming ability. *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education,* Norfolk, Virginia, USA. 210-213.

Dick, M., Sheard, J., Bareiss, C., Carter, J., Joyce, D., Harding, T., et al. (2002). Addressing student cheating: Definitions and solutions. *ITiCSE-WGR '02: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education,* Aarhus, Denmark. 172-184.

Fincher, S., & Petre, M. (2004). *Computer science education research*. London: Routledge Falmer.

Frandsen, G. S., & Schwartzbach, M. I. (2006). A singular choice for multiple choice. *ACM SIGCSE Bulletin, 38*(4), 34-38.

Kumar, A. N. (2003). The effect of closed labs in computer science I: An assessment. *Journal of Computing Sciences in Colleges, 18*(5), 40-48.

Lancaster, T., & Culwin, F. (2004). A comparison of source code plagiarism detection engines. *Computer Science Education, 14*(2), 101-117.

Lister, R., & Leaney, J. (2003). Introductory programming, criterion-referencing, and bloom. *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education,* Reno, Navada, USA. 143-147.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125-180.

Muller, O. (2005b). Pattern oriented instruction and the enhancement of analogical reasoning. *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research,* Seattle, WA, USA. 57-67.

Prior, J. C., & Lister, R. (2004). The backwash effect on SQL skills grading. *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* Leeds, United Kingdom. 32-36.

Ramsden, P. (1992). *Learning to teach in higher education*. London: Routledge.

Roberts, T. S. (2006). The use of multiple choice tests for formative and summative assessment. *ACE '06: Proceedings of the 8th Australian Conference on Computing Education,* Hobart, Australia. 175-180.

Robillard, P. N. (2005). Opportunistic problem solving in software engineering. *IEEE Software, 22*(6), 60-67.

Rowntree, D. (1977). *Assessing students. how shall we know them*. London: Harper & Row.

Stiggins, R. J. (2005). *Student-involved assessment for learning*. Upper Saddle River, NJ: Prentice-Hall.

Tyler, R. W. (1949). *Basic principles of curriculum and instruction*. Chicago: The University of Chicago Press.

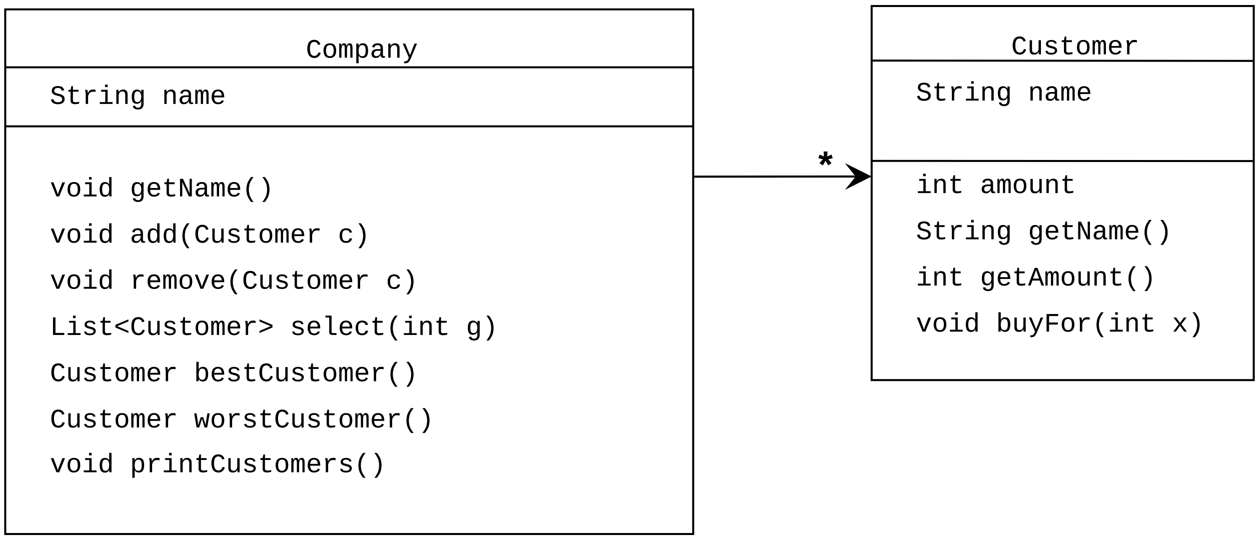
Winslow, L. E. (1996). Programming pedagogy — a psychological overview. *SIGCSE Bulletin, 28*(3), 17-22.

Woodford, K., & Bancroft, P. (2005). Multiple choice questions not considered harmful. *ACE '05: Proceedings of the 7th Australasian Conference on Computing Education,* Newcastle, New South Wales, Australia. 109-116.

## A. Appendix

This exercise is about handling customers in a firm where –among other things – the firm needs to keep track of how many money each customer buys for.

Implement the class model below and a Driver-class that tests all the essential parts of the classes Customer and Company.



Company

In `Company` the field-variable `name` and the methods `getName`, `add` and `remove` are obvious.

The method `select(int x)` must return a list of customers who have bought for a least `x` Kroner.

The method `bestCustomer` must return a customer who has bought for the most amount of money, and likewise must `worstCustomer` return a customer who has bought for the least amount of money.

Note the indefinite article ”a” customer … If more than one customer has bought for the maximum amount of money (respectively the minimum amount) it is secondary what customer will be returned

The method `printCustomers` prints out all customers with one customer pr. line. The printout must be ordered alphabetically by customer-name.

Customer

In the class `Customer, name` is a customers name and `amount` is the total amount a customer has bought for.

The methods `getName` og `getAmount` are obvious. The method `buyFor` is called when the customer have bought for a given amount (x) and must cause `amount` to be updated so that the new amount is reflected in the field variables value..

The method `toString` must return a String-representation of a `Customer`-object in the following form (example): ”Karsten Riis, 7500”.