Transitioning to OOP/Java — A Never Ending Story

Jürgen Börstler, Marie Nordström, Lena Kallin Westin, Jan-Erik Moström, and Johan Eliasson

Department of Computing Science, Umeå University, Sweden
{jubo,marie,kallin,jem,johane}@cs.umu.se

Abstract. Changing the introductory programming course from a traditional imperative model to an object-oriented model is not simply a matter of changing compilers and syntax. It requires a profound change in course materials and teaching approach to be successful. We have been working with this transition for almost ten years and have realized that teaching object-oriented programming is not as simple or "natural" as some proponents claim. In fact, it has proven difficult to convey to the students the advantages and methodologies associated with object-oriented programming. To help ourselves and others in a transition like this we have developed a number of "course design principles" as well as teaching methods and examples that have proven to have positive influence on student learning outcome.

1 Introduction

The object-oriented paradigm has become the most common programming paradigm for introductory programming courses[1][de Raadt et al., 2004; Stephenson and West, 1998; Chen et al., 2005]. The transition to this paradigm has proven to be more difficult than expected. Traditionally, programming concepts have been systematically introduced one after one, each building nicely on the concepts already learned. Abstract and advanced concepts (e.g., modules and abstract data types) were deferred to later courses. In the object-oriented paradigm, on the other hand, the basic concepts are tightly interrelated and cannot easily be taught and learned in isolation [Roberts et al., 2006], as illustrated in Figure 1. Furthermore, the basic object-oriented concepts are on a higher level of abstraction[2]. Together, this results in a higher threshold and steeper learning curve for the learner.

It is generally accepted that transitioning to the object-oriented paradigm is not just a programming language issue. Object-oriented development requires a new way of thinking [Bacvanski and Börstler, 1997]. This is particularly important in education. A syntax-driven approach can take the students' attention

[1] Even in upper secondary school (high school).
[2] Whether this is an advantage or disadvantage for teaching or learning is unclear.

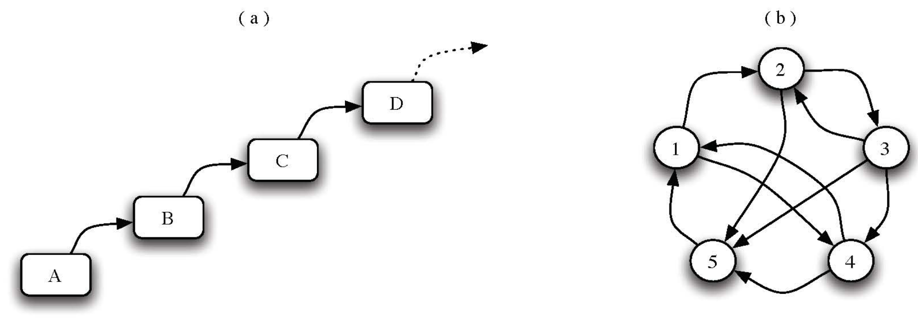

Fig. 1. Dependencies between basic concepts to be introduced in imperative - (a) and object-first approaches (b), respectively

away from the underlying concepts and principles (see also *Model-Driven Programming* by Bennedsen and Caspersen and *CS1: Getting Started* by Caspersen and Christensen). Studies show that there is a mismatch between the programming language used and the paradigm that is actually taught. In Australia for example, about 82% of the introductory programming instructors used an object-oriented language, but only about 37% taught their courses by using an object-oriented approach [de Raadt et al., 2004]. Approaches for teaching introductory programming courses are still heavily discussed [Bruce, 2005].

In this chapter, we describe our experience transitioning from a traditional approach using (Turbo) Pascal to a true objects-first approach in Java. The advantages of using object-orientation for teaching are many. It provides powerful mechanisms for the structuring and organisation of models (in particular designs and code) and decreases the conceptual distance between problem and solution models. This makes it much easier to communicate models and keep them consistent [West, 2004].

However, these advantages come at a price. The basic object-oriented concepts are highly interrelated and cannot easily be taught or learned in isolation (as illustrated in figure 1(b)). There also is no commonly accepted pedagogical approach to overcome this problem [Bruce, 2005]. It is furthermore very difficult, if not impossible, to develop "simple" examples. A proper and meaningful object-oriented example requires quite some overhead [Westfall, 2001]. Many textbook examples are, therefore, unnecessarily complex, not meaningful, or not even "truly" object-oriented [ACM-Forum, 2002; Hu, 2005]. During our "journey from Pascal to Java" we stumbled across these and other problems and made a few detours before realising that the object-oriented approach not only requires a new way of thinking, but also, a new way of teaching.

There are more factors contributing to a student's success or failure than the course material and how a course is taught. We have observed that our students as a group are less motivated and not as well prepared[3] compared to a few

[3] This problem has been observed by most math, science and engineering programs.

years ago. Attendance rates at exams, lectures and other scheduled events have decreased. Reading assignments are neglected to a high degree and mandatory assignments are submitted late. We introduced Supplemental Instruction (SI) [Arendale, 1997] to increase student activity and thereby improve course outcome. SI is a non-mandatory part of the course. In some course offerings, only half of the students participated in the SI programme while in other the majority of students participated. In all course offerings, students participating in SI have a higher attendance rate at exams and also get higher grades on average than the group of students not attending SI [Nordström and Kallin Westin, 2006].

The remainder of this chapter is organised as follows. First, we briefly explain the motives behind the principles we used for designing our new course, and how these were actually implemented. In section 4, we evaluate how well this new course worked with respect to our original principles. Section 5 summarises the lessons we have learned since we made the transition to the object-oriented paradigm in 1998. In sections 6 and 7, we discuss external factors affecting student performance and related work, respectively. The chapter concludes with a summary of our experience.

2 Principles for Course Design

Prior to our transition, we introduced object-oriented concepts in our data types and algorithms[4] course, following the introductory programming course. However, this was not sufficient to enable students to effectively use the object-oriented paradigm. Most students perceived object-orientation as a simple extension to imperative programming. They did not realise that object-oriented programs are conceptually different from strictly imperative ones and that using object-oriented syntax does not automatically lead to object-oriented programs.

When switching to the object-oriented paradigm in our introductory programming course in 1998, we only made minor changes to our traditional course design. Initially, our students did very well on this course, but we soon realised that their ability to develop code true to the object-oriented paradigm was not satisfactory. After several course offerings with unsatisfactory learning outcomes, we decided to develop a "truly" objects-early approach. When proficiency in a certain paradigm is the major learning goal of a course, it seemed sensible to start with that paradigm as early as possible [Bruce, 2005; Bergin, 2000b].

To support the design of such a course we developed a list of principles, to guide course development. These principles were either based on our teaching experience [Bacvanski and Börstler, 1997; Börstler et al., 2002; Börstler and Sharp, 2003; Kallin Westin and Nordström, 2003, 2004] or the collected advice and experience from the literature in computer science education (see e.g., [Bruce, 2005; Westfall, 2001; ACM-Forum, 2002; Guzdial, 1995; Holland et al., 1997; Kölling and Rosenberg, 2001; Kölling, 2003; Turk, 1997; and *Using BlueJ to Introduce Programming* by Kölling]).

[4] More or less similar to a CS2 course.

2.1 High-Level Goals

No magic (P1). We must provide a correct and consistent frame of reference, so that the students always can make sense of new material. The students must be able to associate the new material with something familiar or wellknown. The succession of learning units and topics must be carefully worked out. The frame of reference must be refined or extended accordingly. The current frame of reference should always be sufficient to understand new material and validly explain what is going on [Zull, 2002]. Everything requiring a comment like “don't worry now, you'll understand later,” must be revised or delayed. Language specific complexities should be hidden until students are sufficiently mature to understand the underlying language design issues.

Students will always try to make sense of new material. If we cannot provide them with a correct and consistent frame of reference, they might construct invalid explanations by themselves. This can easily lead to persistent misconceptions about object technology and programming [Holland et al., 1997; Börstler, 2005; Clancey, 2004; Ragonis and Ben-Ari, 2005].

Objects from the very beginning (P2). Everything should build on the notion of objects, since they are at the very heart of object-orientation. Therefore, objects should be introduced in the very first lecture. The earlier we start with the most important concept, the more often we can reinforce it and the more time we give students to fully understand it.

General concepts favoured over language specific realisations (P3). Learning units should be based on the teaching and learning of general object-oriented concepts. Although the mastery of a particular programming language is an important learning goal, it is secondary to the understanding of the underlying concepts. Focusing on concepts does not necessarily mean to move strictly from concept to syntax for each new topic. It is, however, important to stress fundamental principles and techniques and not the elements of a particular language. This can, for example, be achieved by means of moving from concrete to abstract as proposed in *CS1: Getting Started* by Caspersen and Christensen.

No exceptions to general rules (P4). By general rules, we not only mean the definitions that constitute the object-oriented paradigm[5], but also design and programming guidelines and all the other pieces of advice we provide to our students. We must always “do as we say,” only use sound and meaningful objects, only show well-designed classes, and certainly not do unnecessary `main`-programming. Concepts must never be introduced or be reinforced by using flawed examples (see also P6).

2.2 Tools

OOA&D early (P5). It is necessary to provide students with simple tools to approach a problem systematically and to evaluate alternative solutions before

[5] Like for example “objects are instances of classes with state, behaviour and identity,” or “in object-oriented programs, problems are solved by objects sending messages to each other.”

starting to code. Early OOA&D conveys to the students that responsibilities are distributed amongst the objects that solve a problem [Börstler, 2005; Andrianoff and Levine, 2002.]

Exemplary examples (P6). All examples used in classes and exercises should comprise well-designed classes that fill a purpose (besides exemplifying a certain concept or language specific detail) [Holland et al., 1997; Nordström, 2007]. Consequently, examples should be non-trivial and involve multiple classes. All examples should be made available for experimentation, e.g., by making the source code available for download from the course web page.

Easy-to-use tools (P7). Students should be provided with tools that support object-oriented thinking. The tools must be easy to learn and easy to use. Tool usage must add as little cognitive load as possible to the students' tasks. Usability should be favoured over any "bells and whistles."

2.3 Pragmatics

Hands-on (P8). Programming is a skill that must be "trained." Topics should be reinforced by means of practical exercises. Lectures should be followed by supervised in-lab sessions. For each session, step-by-step instructions and exemplary examples should be provided.

Less "from scratch" development (P9). "Reading before modifying before coding." All software development takes place in context (see also *Using BlueJ to Introduce Programming* by Kölling). Reuse is an important aspect of the object-oriented paradigm and should be emphasised early. To be able to read, to understand, and to modify existing code is, therefore, as important as developing understandable code.

Alternative forms of examination (P10). Assessment should support learning. It is very important to evaluate actual programming skills as well as conceptual understanding. Furthermore, assessment should not be separated from teaching. For example, peer evaluation or peer marking can call the students' attention to alternative ways of solving certain problems. It is important to realise that there rarely is a single, correct solution to a problem.

Emphasise the limitations of computers (P11). Students should learn that computations can produce erroneous or unexpected results due to limitations in data representation, even in logically correct programs.

To summarise, our main goal was to follow an object-oriented approach in a true and consequent way. In addition, we wanted to provide our students with easy-to-use tools supporting "object thinking" and the systematic development of proper object-oriented code (see also *Model-Driven Progamming* by Bennedsen and Caspersen).

3 Implementation

The first course, based on these principles, was offered in spring 2001. After a case study in summer 2001 [Börstler et al., 2002], we have refined our teaching approach and successively implemented it in all our introductory programming courses[6]. Some of the principles are very difficult to implement, or even in conflict with each other. In particular the principles *No magic (P1)* and *Exemplary examples (P6)* still cause a lot of work.

From an organisational point of view, we made four major changes to our original course as follows:

1. We introduced BlueJ [BlueJ, 2007], a programming environment particularly designed for the teaching and learning of object-oriented programming to novices (P7) [Kölling and Rosenberg, 2001; Kölling et al., 2003]
2. We introduced CRC-cards, a simple informal tool for collaborative object-oriented modelling (P5, P7) [Beck and Cunningham, 1989]. The strength of the CRC-card approach lies in its associated scenario role-play activities [Börstler, 2005; Andrianoff and Levine, 2002]. During the role-plays the students explore hypothetical, but concrete situations of system usage (scenarios). They enact the objects in the model, much like actors following a script when playing the characters in play. This supports "object thinking" and helps the students to develop a mental model of the workings of an object-oriented program (P1-P4) [Börstler and Schulte, 2005].
3. To accommodate for more practical training (P8), we substituted our traditional lecture room exercises by guided, hands-on exercises in computer-labs.
4. The traditional pen and paper exam was split into a shorter one, half way through the course, and a computer-based exam was used at the end to test actual programming skills (P10).

In addition to these changes, we started to offer Supplemental Instruction (SI) [Kallin Westin and Nordström, 2003] to improve students' study skills and to make them more active participants in the course. SI is targeted towards historically difficult classes to help students master content while developing and integrating strategies for learning and studying [Arendale, 1997]. This is done through sessions guided by a model-student, the SI leader.

A major difference between SI and other forms of collaborative learning is the role of the SI leader. Rather than forming study cluster groups and then releasing them in an unsupervised environment, the SI leader is present to keep the group on task with the content material and to model appropriate learning strategies that the other students can adopt and use in the present course, as well as other ones in future academic work.

Since the "roll out" of our teaching approach, we have made several changes to our introductory programming courses (see Figure 2 for an overview). For example, we have postponed graphics and event handling to a newly developed advanced programming course. We have also slightly adapted our course for non-CS majors. However, we are still faithful to all our principles.

[6] We offer introductory courses in object-oriented programming for three different technical degree programs.

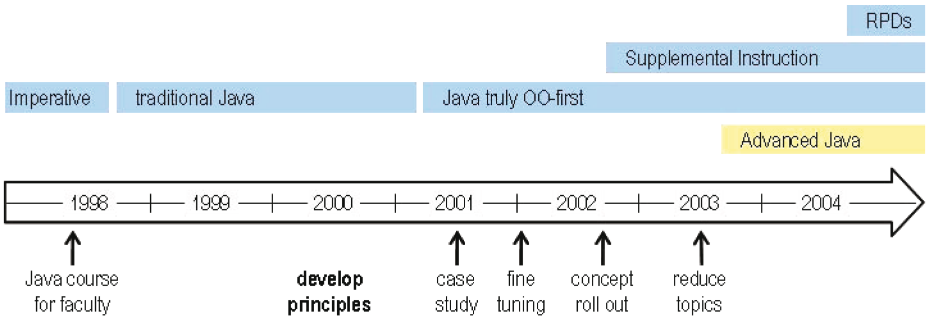

Fig. 2. Major steps in the evolution of the introductory programming course

4 Evaluation

In this section, we restrict our discussion to an evaluation of our principles as defined in section 2. Overall, we conclude that changing to a "truly" object-oriented approach according to our principles worked well. However, there are many factors not directly related to the teaching and learning of object-oriented programming itself that affect course design and outcome. Many important factors are difficult to control like prerequisites and attitudes of the students entering our programs, for example. This will be discussed in section 6.

4.1 High-Level Goals

In common for all high-level goals (P1-P4) is the urge to be "truly faithful" to the object-oriented approach. This means to avoid concepts or examples that seem to question or even contradict the idea of object-orientation, such as objects without meaningful state or behaviour, excessive use of static methods and public attributes, Singletons[7], etc. [Westfall, 2001; Hu, 2005]. To be "truly faithful" also means to strive for meaningful objects in realistic contexts.

No magic (P1). Our ambition has always been to use examples and contexts not only simple enough for the students to understand, but that also emphasises the object-oriented paradigm. BlueJ is an excellent tool for this since it allows teachers and students to concentrate on the object-oriented aspects instead of dealing with editors, configuration files, compilers, etc. BlueJ achieves this by visually representing classes and objects and manipulating them directly using its graphical user interface (see figure 3). Unfortunately, this approach has some limitations that can generate misconceptions that can be harmful and great care has to be taken to avoid them. A short example will illustrate the problem.

Since the very beginning we have used an example with geometrical shapes supplied with the BlueJ environment [Barnes and Kölling, 2003; and *Using BlueJ to Introduce Programming* by Kölling]. In BlueJ, objects are represented by red

[7] A Singleton is a class with one single instance only.

blocks in the object bench (see for example c: Circle in figure 3). Actually, to be more precise, these red blocks represent object references and not the objects themselves. This is a small, but important difference as explained below.

One can send messages to objects in the object bench by right clicking them. This will display a menu with the methods defined for this object. When selecting `makeVisible()`, a graphical surface is created ("automagically") and a representation of `c` is drawn on it (see Window `BlueJ Shapes Demo` in Figure 3). Whenever the state of an object is changed, its representation is changed or animated accordingly. This gives immediate feedback and helps students to understand the difference between classes and objects. On the other hand, it blurs the difference between the objects themselves and their references. Furthermore, the details of the graphics are quite involved and too complicated to understand ("magic") for a novice.

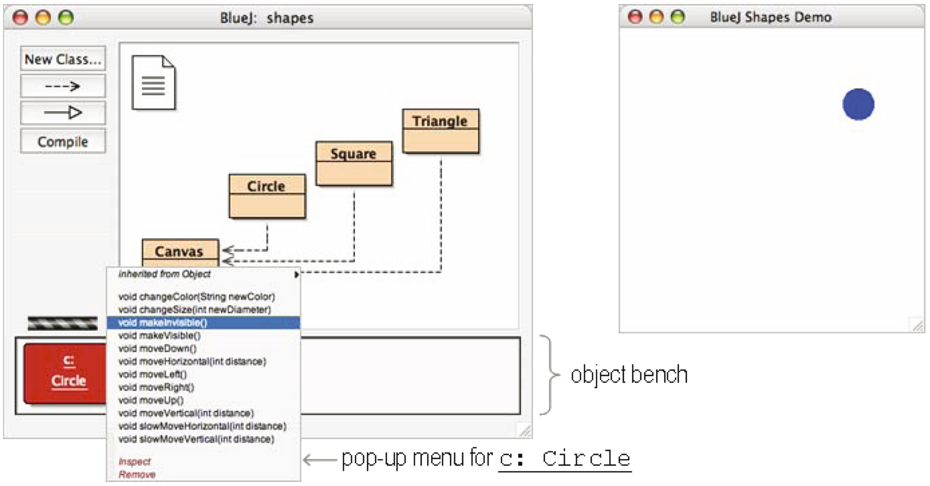

Fig. 3. Screenshot of the Shapes example

Another problem with the Shapes example is the cognitive difficulty to differ between the visual representation of an object (the circle in window `BlueJ Shapes Demo`) and the object itself, which actually cannot be seen. If, for instance, the reference to the object (the red block in the object bench) is removed, nothing happens in the drawing. This is puzzling for the inexperienced because the coloured dots on the canvas are mistaken for the object itself! Misconceptions like this are very hard to deal with. Other examples of difficulties are discussed in [Ragonis and Ben-Ari, 2005].

This example shows how difficult it is to create assignments early in the course, without (unintentionally) introducing magic or material not taught yet.

Objects from the very beginning (P2). To make the students immediately acquainted with the idea of objects, we use a kind of interactive exercise the first lecture [Andrianoff and Levine, 2002; Bergin, 2000a]. Without previous explanation, the students are asked to discuss in general terms, something that needs to be modelled like, for example, a ticket machine or an employee. During the discussion the lecturer collects specific and general characteristics and behaviours on the whiteboard. At the end of the lecture, these things are pointed out as "properties" belonging to a single object or a class, respectively.

Nevertheless, it is difficult to convey to the students that they are working with an isolated "component" in a larger program, instead of a whole program. Many students, particularly those with previous programming experience, find it quite frustrating that there is no "program" to execute like they are used to. They seem to have difficulties focusing on the properties and responsibilities of objects without controlling its context at the same time [Guzdial, 1995].

General concepts favoured over language specific realisations (P3). The learned programming and problem solving should be transferable to other programming languages. It is important, therefore, to focus on general concepts. We try to highlight general concepts, knowledge and skills and to avoid language specific details and idiosyncrasies.

This has resulted in using elementary UML-notation throughout the course, instead of some kind of simplified temporary notation. However, we do not explicitly introduce UML. We just use its most intuitive parts consistently.

Furthermore, semi-formal syntax-diagrams are used. This makes it much easier to talk about the syntax, semantics and pragmatics of programming languages. Information hiding is also stressed as a general (design) concept and the usefulness of Boolean variables to formulate easy to read expressions.

On the other hand, topics like anonymous objects and classes are not discussed. These concepts require a thorough understanding of object-orientation and are saved for later courses. We try to avoid language idiosyncrasies as long as possible in particular shortcuts like ++, +=, ?:, etc. and forced returns out of for-loops and methods. They just make code harder to read and, therefore, their use is actually discouraged.

No exceptions to general rules (P4). It is important to be consistent with the frame of reference we provide to our students (see P1 in section 2.1). Students will hopefully adopt what the teachers present to them eventually. It is important, therefore, not to misguide them (see also P6). We must never present any material, explanation or example that we might reject as an answer or solution from a student.

Unfortunately, Java courseware in particular is littered with examples that contradict the "rules" or "styles" we want our students to adopt. The concept of objects, for example, should not be exemplified by using Java strings. In Java, `String` objects cannot be modified and do not posses all the characteristics we require from proper objects [Thimbleby, 1999]. Since `Math` has only `static` methods and there are no objects of this class type, its use should be postponed until the students have a firm understanding of the concepts class and object. The `main` method is an atypical method since there is no object it belongs to.

Thus, the method is never invoked explicitly and its parameters seem to be supplied by magic forces (see also P1).

4.2 Tools

OOA&D early (P5). The purpose of this principle is twofold: showing the students a systematic way to develop a solution for a given problem, and providing them with a tool to reason about object-oriented solutions without the need of actual code.

By using CRC-cards [Beck and Cunningham, 1989], we can do both. The object-as-person metaphor helps students with "object thinking" and to develop a conceptual model of the inner workings of an object-oriented program [West, 2004; Börstler and Schulte, 2005]. Another advantage of this approach is that it does not require any prerequisite knowledge.

A problem noted in [Bellin and Simone, 1997] and described in detail in [Börstler, 2005] is that CRC-cards are used as surrogates for classes (in the modelling activities) as well as for objects (in the role-play activities). This conflicts with the *No exception* principle (P4) and can easily confuse novices.

To address these problems, we enact a live CRC session in front of the class to give the students a feeling for the dynamics of such a session. In addition to that, we have developed so-called Role-Play Diagrams (RPDs) to support and document the role-play activities [Börstler, 2004]. RPDs combine elements from UML object and collaboration diagrams [OMG, 2003]. However, the notation is informaland much less extensive. In RPDs, we use specific object cards to denote objects and thereby, avoid the double role of the CRC-cards. Although the enhanced "method" is more complicated than the original one, the students have fewer problems using it. The RPDs also provide an excellent documentation of the role-play. To give the students some practical experience in CRC-card role-playing, we schedule two CRC exercises where the students develop designs for small problems. One of these designs is later implemented as an assignment.

Exemplary examples (P6). As discussed in *No magic* (P1), it has turned out to be difficult to find or to develop suitable problems and examples for the initial introduction of objects. The range of concepts and syntactical elements known to the students is still very limited. Examples should also be small and to the point, so that students do not lose sight of the concept exemplified. This limits the degree of freedom for defining "realistic" objects. For example, what would constitute a reasonable context illustrating the concepts of loops? What kind of object would have such behaviour? Immediately the example grows to justify the use of a simple construct and tends to conceal the small component it was intended to show.

Another problematic example is the usage of Singleton classes, like the popular Pig-Latin translator [Nordström, 2007]. One might ask whether it is reasonable to have a class `PigLatinTranslator`? How many objects of this class would anyone need? Singleton classes do probably not qualify as good examples. The main idea behind classes is instantiating as many objects as necessary. Singletons are special cases, i.e. an exception to the general rules (c.f. P4). Their treatment should, therefore, be delayed to more advanced courses.

Many examples use print statements to present some result. This is not a representative way to illustrate objects and classes. Usually, results are returned and used by other objects. In an object-oriented program, objects communicate to fulfil a task. Objects that use printing to present results are rarely useful in other contexts. Students are not able to reuse such examples as prototypes or templates to solve more general problems.

Exploiting the "naturalness" of the object-oriented approach can also be difficult. Object-oriented models of real-life objects might have behaviours and responsibilities their real life counterparts never would or could have. Therefore, it is very important to make a distinction between the model and the entity being modelled. A typical example of this could be the model of an employee in an economy system for a company. The model of the employee could have the responsibility to know its salary, the number of remaining days of vacation and so on. This is conflicting to how things are in real life. No company would rely on their employees to be the only source of information for the payment of salaries. So, how could it be possible for the inexperienced designer to foresee this responsibility in the model?

Easy-to-use tools (P7). Some of the advantages of BlueJ turned out to be disadvantages for the students (initially). The interaction with entities is done by right-clicking a class or an object. The problem for the student is to understand the equivalence of right-clicking and generating the same action in code. Another problem is to realise the difference between classes and objects [Ragonis and Ben-Ari, 2005]. However, as the students continue to practise their skills using BlueJ they realise the strengths of this simple, but powerful, interaction with objects.

The ability to write code must not depend on the tools we provide to our students. Students must not be "locked" into BlueJ for example. This is also highlighted by the BlueJ developers (see also *Using BlueJ to Introduce Programming* by Kölling). They should develop and run at least one complete application outside BlueJ. Although experienced students tend to dislike BlueJ, we think they should be encouraged to at least try it. They might very well get some new insights into the object-oriented paradigm.

4.3 Pragmatics

Hands-on (P8). The initial idea of guided in-lab sessions directly following the lectures did not work as expected. The students complained about lack of time to think about the new material before using it. Most students actually had difficulties applying the ideas presented. They merely consumed the presentation at the lecture.

In later years, we have thus rescheduled the lab sessions. We still have the same number of hands-on sessions, but they are no longer scheduled on the same day as the corresponding lectures. We also developed very detailed guides to make sure students succeed with initial tasks and so they can gain some confidence in working with the environment. Too detailed guidelines or fill-in-the blanks exercises, however, can be counter-effective. Students might be enabled to perform successfully without actually understanding their answers and activities. Students and teachers as well might get a faulty feeling of mastery of the subject.

Less "from scratch" development (P9). The practise of reading and manipulating existing code before actually writing own code turned out to be a major problem for our students. Inexperienced students acquired a passive practice and had difficulties writing complete programs on their own. It is important then to train some programming from scratch. Experienced students, on the other hand, often want to have full control over their programs and might reject "foreign" code [Guzdial, 1995]. However, code reuse is a crucial practice that requires code reading and understanding and needs to be trained as well.

Alternative forms of examination (P10). The content of the course is initially focused on object-oriented concepts, while the second half is heavier on actual problem solving and programming. To reinforce the need to work with and to understand basic concepts early on, we divided the examination into two parts. Halfway through the course a written (theoretical), closed-book test is given and at the very end, a practical problem solving and programming test is given. The results of the two tests are added and graded as one. In addition to this, the students have mandatory assignments and some of them orally assessed. The idea of splitting the examination into two tests with rather different focus was appreciated by the students. Furthermore, the exam results better reflect student skills than a single pen-and-paper test.

Emphasise the limitations of computers (P11). This principle had its origin in the numerical tradition of our department. We make students aware of problems and limitations in data representation and how these can lead to erroneous computations. We emphasise this by discussing examples leading to unexpected results in logically correct programs.

5 Lessons Learned

In this section, we briefly summarise the practices that worked particularly well for us. We have grouped them together into recommendations to make them easily accessible to the reader.

Teach "object thinking" and modelling explicitly.

– Start the first lecture with a modelling or role-play activity (no syntax involved). Students can be asked then to describe (model) an employee or a ticket-machine to illustrate the basic object properties (state, behaviour and identity).
– Introduce CRC-cards and scenario role-plays. This provides students with a framework to think in terms of (active) objects and their responsibilities. Furthermore, it teaches them basic modelling/ problem solving skills.

- Introduce role-play diagrams so that students easily can track and document scenario role-plays. This also helps to prevent some problems inherent in the original CRC approach [Börstler, 2005; Börstler and Schulte, 2005].

Schedule guided and supervised lab activities. Programming is a skill that needs to be trained extensively. Students should visit the labs as frequently as possible and receive immediate help when getting "stuck."

- Reduce the number of traditional lectures and introduce supervised lab sessions instead. Guide students through practical exercises in the labs. We provide for example step-by-step guides, including reflective questions, which the students have to work through. Lecturers and teaching assistants should always be present to discuss and resolve problems.
- As much as possible, move supervising time from office rooms to the computer labs to force students to visit the labs to ask questions.

Use and utilise a suitable programming environment. The environment must be easy to use and to support the object-oriented paradigm. However, it is also important to show how programs are developed and executed outside such an environment. We have used BlueJ [BlueJ, 2007] successfully since 2001.

Examine the "right" things. It is important to assess actual and individual programming skills in addition to conceptual and syntactical knowledge. This can be done, for example, by practical computer based tests (problem-solving and programming) and individual oral demonstrations of mandatory assignments.

There is no course design that fits all target groups. Different groups of students need different types or flavours of courses. It is important to be sensitive to changes in the field as well as the context and the environment of a course [Forte and Guzdial, 2005; Jenkins and Davy, 2000; and *Using On-line Tutorials in Introductory IT Courses* by Thomsen]. Our principles have been a useful guideline to us when adopting the course to different student groups. The principles make sure that the core of the course is the same and taught in roughly the same way, regardless of lecturer and student group.

Do not lull students and teachers into false security. Fill-in-the-blanks guides and exercises can give a faulty feeling of students' subject mastery. Too much help or undemanding tasks can lead to mechanical answering. If no reflection or second thought is necessary, then students can successfully complete such exercises without learning anything. Also, teacher expectations about what the students really have learnt might be too high.

Good examples are crucial, but very hard to develop. Truly object-oriented examples are very difficult to find or to develop. Educators should resist constructing examples "on-the-fly" (for example to exemplify a specific feature), since they rarely will follow principles P1, P4 and P6.

– Programming in a true object-oriented style often leads to overly (unnecessary) complex examples, due to the additional layers of abstraction imposed by the paradigm. This can be frustrating to students since they cannot understand why the different abstraction layers are necessary (e.g., "Why should I do it like that, it's easier and faster to read the information directly from the database"). It is a challenge for the teacher to explain that optimization is secondary to a good object-oriented design. Our main goal is to devise a good solution that fulfils certain quality criteria and not to simply make it work somehow. Students are not mature enough to differ between optimizations and proper design.
– Although often claimed, there is no 1:1 relationship between real-world objects and their corresponding software abstractions. A physical book for example is removed from the library, when it is checked out. A book object in a (software) model, however, stays in the library and is only marked as "on loan." Furthermore, in a "real world" library, we would never make the borrowers responsible for keeping track of their unpaid overdue fines. In a (software) model however, this might be a good design choice since trust is no issue there.

Keep students active. Data collected during the SI-projects shows without a doubt that student attendance and activity correlate with course results [Arendale, 1997; Kallin Westin and Nordström, 2003, 2004.] Mandatory in-lab exercises and a two-stage examination keep the students alert and active from the start. SI gives the students opportunities to work with the course material in a structured way and helps them to recognise the strength of collaboration. After introducing SI, the attendance rate on the examination rose from 80 percent to above 95 percent (see Section 6, in particular Figure 4).

6 Discussion

When analysing student performance over the years (see Figures 4 - 6) our case for a "truly-objects-first" approach does not look convincing. However, there are also external factors affecting student performance. These factors have lead to considerable changes in the student population in recent years.

Assessment consists of mandatory assignments, a pen-and-paper test and a computer-based test (see P10 in Section 4.3). In Figure 4, the passing rate after the first opportunity to finish the course is shown as a solid line. Java was introduced in 1998 and our "truly" object-first approach was introduced in 2002 (see Figure 2). The dashed line in Figure 4 shows the attendance rate on the exam (i.e. the proportion of students submitting at least one mandatory assignment or attempting at least one test). SI was introduced in 2002 to raise attendance rates and it seems to have an effect[8]. Participation in SI correlates with overall student

[8] In 2001, the seemingly high attendance rate was due to an examination system where handing in assignments (not necessary correct ones) gave credit points on the final exam. The numbers for 2001 in figures 4 - 6 must be seen, therefore, as statistical outliers.

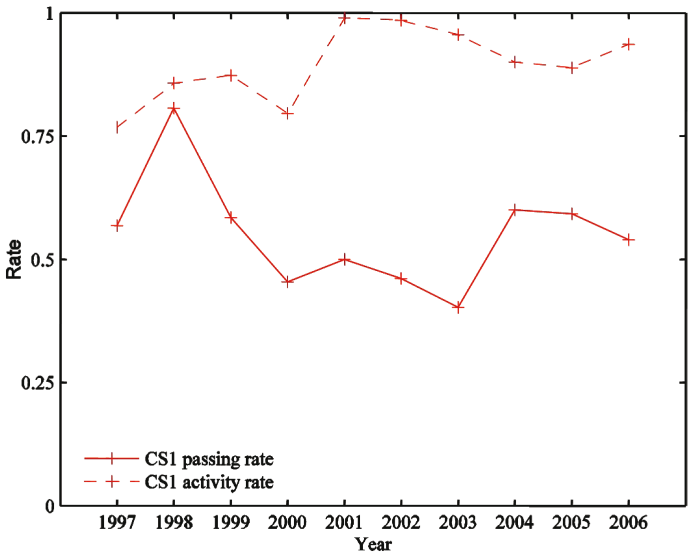

Fig. 4. Performance data for CS majors on our introductory programming course. The solid line represents the passing rate after the first opportunity to finish the course. The dashed line represents the proportion of students submitting at least one mandatory assignment or attempting at least one test.

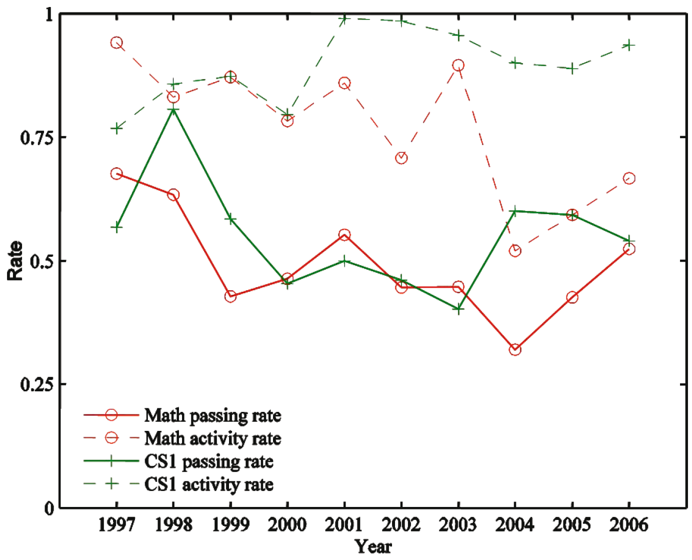

Fig. 5. Performance data for CS majors on the discrete mathematics course, compared to our introductory programming course (cf. figure 4)

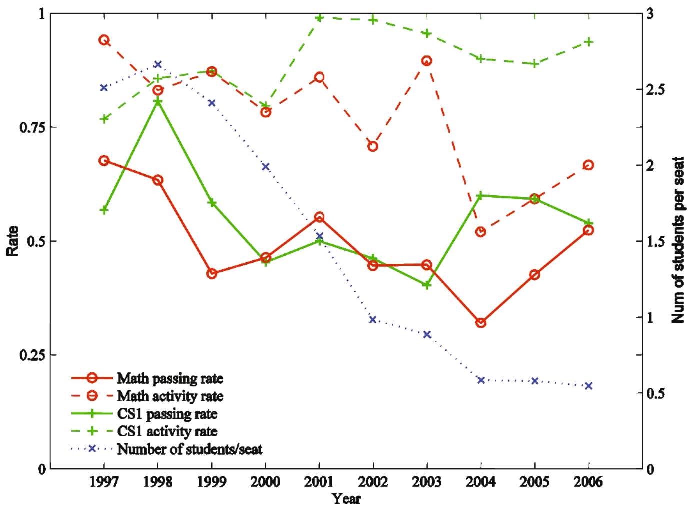

Fig. 6. The number of students per seat on our programme is shown as a dotted line added to Figure 5

performance. Unfortunately, the weakest students seem not to be motivated to participate in SI. An investigation of the students with severe problems in keeping up with the pace of the curricula of the programme showed that a vast majority had not attended SI at all, or only tried it a few times.

Another factor is that knowledge and skills in mathematics have been decreasing in general [Högskoleverket, 1999; Helenius and Tengstrand, 2005]. It is believed that mathematical ability is strongly connected to performance in introductory programming [Denning, 2004]. Skills, like (array) indexing and creating series of numbers seem to be more of a problem nowadays. Students also have a weak understanding of functions, in particular, their parameters and return (computed) values. This lack of understanding might result in the assumption that the only possible way to get something out of a method is by printing a string to the screen. We strongly believe this contributes to the lower passing rates, especially for mandatory assignments.

Our CS majors take a course in discrete mathematics in the same term as their introductory programming course. In Figure 5, we can see that the students also have problems in the discrete mathematics course. Attendance rates are even lower in discrete mathematics where students only have a single traditional exam at the end of the course. However, in the introductory programming course, a student needs to attend only one of the three exam parts to be counted as "active". This might falsely indicate high attendance rates.

The dotted line in Figure 6 represents the number of students per seat. In Sweden, each programme has a fixed number of student seats available. The applications to IT-related programmes have severely suffered from the turbulent situation within the IT business. Practically, this means that all students applying are admitted as long as they fulfil the basic prerequisites. Historically, we have had 2 to 3 applications for each seat available, resulting in a higher grade average for the students admitted.

A further factor is a shift in motivation among novices. In a study we performed in 1994, the main reason for students applying for our programme was an interest in the subject itself (or in mathematics). In a later study, the motivation had shifted to "want high salary," "want to be a civil engineer," and other reasons not connected to the subject or the programme itself. Thus, students' interest in computing science is far from obvious [Kallin Westin and Nordström, 2001, 2003; Eliasson et al., 2006a,b]. Similar trends are reported internationally [Forte and Guzdial, 2005; Jenkins and Davy, 2000].

7 Related Work

There have been several attempts to explain why students are having difficulties in their first Java course. Three common explanations are the following:

– The students can program, they are just having problems with the design part [McCracken et al., 2001].
– We are not teaching object-orientation the correct way, we need to teach the subject in a pure, object-oriented way [Bergin, 2000a; Kölling and Rosenberg, 2001.]
– Java has so many special cases, like `public static void main` and string handling so that it becomes difficult for the students to remember and to understand all the special cases [Bruce et al., 2005].

In a multi-national cross-university study, [McCracken et al., 2001] investigated how well students actually could program. They proposed a list of five steps that students should be able to follow successfully after passing CS1.

– Abstract the problem from its description.
– Generate sub-problems.
– Transform sub-problems into sub-solutions.
– Recompose the sub-solutions into a working program.
– Evaluate and iterate.

The results from this study were disappointing. The students' programming skills were at a much lower level than expected. Somewhat surprising, the most difficult part seemed to be the first step (e.g., to abstract the problem).

[Lister, 2004] followed up on these results and investigated students' ability to read, to understand and to modify existing code. Here, the results were disappointing also. A surprisingly large proportion of the students had difficulties completing even the most basic tasks. It seems that students not only have problems with the abstraction step, they also have problems with the more basic task of reading and understanding code.

[Lister, 2004] also investigated the annotations students made while solving the problems. In general, it turns out that students who carefully trace executions are more likely to provide correct answers than those who do not. However, there are considerable differences between universities [McCartney et al., 2005]. Students from some universities used annotations (traces) to a very high degree while others did not.

Considering the scope of this book it is interesting to note that the two universities with the least annotations are in Sweden and Denmark. Despite the low annotation level, these students performed on average compared to the students from other universities. Whether this is a coincidence or due to differences in object-oriented programming education needs to be further investigated.

8 Summary and Conclusions

The transition to object-orientation is not easy. It is not sufficient to simply change the language of instruction in an otherwise traditional introductory programming course. The strong relationships between basic, objected-oriented concepts constitute a high threshold to the learning and teaching of programming. Considerable changes to the course design are necessary to convey to the students the real power of the object-oriented approach.

We have presented and evaluated a set of eleven principles for course design that have helped us to stay on track in our efforts continuously to improve our introductory programming course. We have seen several factors that influence the results of an introductory programming course apart from the course itself. Attendance rates drop on all parts of the course and many students seem to think that knowledge can be acquired passively.

It is our firm believe that it is necessary to be faithful to the object-oriented approach. Tools that help students to "think in objects" are very important for successfully teaching basic object-oriented concepts. We must provide our students with a consistent frame of reference. This frame of reference will change with the knowledge and skills the students acquire. However, its core (i.e., the basic rules) should not be constantly contradicted by our own exercises and examples.