Programming in Context
– A Model-First Approach to CS1

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

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

ABSTRACT
The recommendations of the Joint Task Force on Computing Curricula 2001 encompass suggestions for an object-first introductory programming course. We have identified conceptual modeling as a lacking perspective in the suggestions for CS1. Conceptual modeling is the defining characteristic of object-orientation and provides a unifying perspective and a pedagogical approach focusing upon the modelling aspects of object-orientation. Reinforcing conceptual modelling as a basis for CS1 provides an appealing course structure based on core elements from a conceptual framework for object-orientation as well as a systematic approach to programming; both of these are a big help to newcomers. The approach has a very positive impact on the number of students passing the course.

Categories and Subject Descriptors
D1.5 [**Programming Techniques**]: Object-Oriented Programming.

D3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and objects*.

K3.2 [**Computers & Education**]: Computer and Information Science Education – *Computer science education, Information science education*.

General Terms
Algorithms, Design, Documentation, Human Factors, Languages.

Keywords
CS1, Conceptual Modelling, Design, Objects-First, Pedagogy, Programming Education, Systematic Programming, UML.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'04, March 3–7, 2004, Norfolk, Virginia, USA.
Copyright 2004 ACM 1-58113-798-2/04/0003…$5.00.

1. INTRODUCTION
Over the years there have been ongoing discussions on the content of an introductory programming course. In order to define a common curriculum including an introductory course, ACM and IEEE established the Joint Task Force on Computing Curricula 2001. The charter was: "To review the Joint ACM and IEEE CS Computing Curricula 1991 and develop a revised and enhanced version for the year 2001 that will match the latest developments of computing technologies in the past decade and endure through the next decade". In the final report [16], the role and place of programming in the curriculum is discussed. Is programming what needs to be taught first (what they call a programming-first approach) or are there other topics that need attention first? The conclusion is: "the programming-first model is likely to remain dominant for the foreseeable future".

The report describes three implementations of a programming-first curriculum based on three programming paradigms: The imperative, the functional and the object-oriented paradigm. The object-oriented paradigm has gained much interest in the past decade resulting in many textbooks (e.g. [2, 3, 10, 12, 15]) and much interest among teachers on implementing the object-first strategy (e.g. [1, 7]).

In [10] three perspectives on the role of a programming language are described:

- *Instructing the computer*: The programming language is viewed as a high-level machine language. The focus is on aspects of program execution such as storage layout, control flow and persistence. In the following we also refer to this perspective as *coding*.
- *Managing the program description*: The programming language is used for an overview and understanding of the entire program. The focus is on aspects such as visibility, encapsulation, modularity, separate compilation.
- *Conceptual modelling*: The programming language is used for expressing concepts and structures. The focus is on constructs for describing concepts and phenomena.

These represent a widespread three-level perspective on object-oriented programming as represented by the three abstraction levels for the interpretation of UML class models [9]: conceptual level, specification level and code/implementation level.

When designing a programming course one decides how much time, effort and focus are given to each of the three perspectives. It is possible just to focus on the first, instructing the computer, and ignore the two others. This results in a course where the details of the programming language are in focus but where the students do not learn the underlying programming paradigm. If on the other hand one just focuses on conceptual modeling (using a case-tool to generate code), the result is a course where the students cannot produce code by themselves. We find it vital to balance the three views on the role of the programming language by including conceptual modeling. The primary advantages are

- A systematic approach to programming
- A deeper understanding of the programming process
- Focus on general programming concepts instead of language constructs in a particular programming language.

Most of the descriptions and discussions of the object-first strategy tend to focus on *instructing the computer* and *managing the program description*. To our knowledge, no introductory programming textbook exists that includes conceptual modeling, and we have been able to find only one article discussing the adoption of conceptual modeling in CS1 [1]. It is our experience from many years of teaching CS1 that including conceptual modeling perspective has a great impact on the student's skills and their understanding of the programming process. It is our firm conviction that the general omission of conceptual modeling is the major reason for the problems identified in [16, p. 23]:

> *Introductory programming courses often oversimplify the programming process to make it accessible to beginning students, giving too little weight to design, analysis, and testing relative to the conceptually simpler process of coding. Thus, the superficial impression students take from their mastery of programming skills masks fundamental shortcomings that will limit their ability to adapt to different kinds of problems and problem-solving contexts in the future.*

[16] generally ignores conceptual modeling in the object-first recommendations for CS1. Aspects of conceptual modeling are mentioned only briefly and the recommended time to be used on the subject is four core hours!

2. CONCEPTUAL MODELING

In [13] object-oriented programming is defined as follows:

> *A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world.*

The key point here is model. An object-oriented program is a model, and this model can be viewed at different levels of detail characterized by different degrees of formality: An informal conceptual model describing key concepts from the problem domain and their relations, a more detailed class model giving a more detailed overview of the solution, and the actual implementation in an object-oriented programming language.

Object-orientation has a strong conceptual framework (notions of concepts and phenomena, identification of objects, identification of classes, classification, generalization and specialization, multiple classification, reference- and part-of composition). One of the advantages of the conceptual framework is that it gives an integrating perspective on analysis, design and programming thus making it much easier for the students to understand these normally fuzzy concepts. Analysis is that process by which you create a conceptual model of the problem domain, design is that by which you fit the model to the restrictions of the particular programming language, and implementation is that by which you implement the design model. Omitting this integrating perspective and focusing only on object-orientation for implementation will leave out one of the most important assets of object-orientation.

We focus on the conceptual modeling perspective, emphasizing that object-orientation is not merely a bag of solutions and technology, but a way to understand, describe and communicate about a problem domain and a concrete implementation of that domain.

The integration of conceptual modeling and coding provides structure, traceability and a systematic approach to program development which strongly motivates and supports the students in their understanding and practice of the programming process.

3. STRUCTURE OF A MODEL-FIRST COURSE

The approach taken here is to use the three perspectives on the role of the programming language as a guide for the structure of the course. In the first half of the course, roughly speaking, focus is concurrently on conceptual modeling and coding; in the second half of the course the primary focus is on internal software quality, i.e. managing the program description.

Coding and conceptual modeling is done hand-in-hand, with the latter leading the way. Introduction of the different language constructs are subordinate to the needs for implementing a given concept in the conceptual framework. After introducing a concept from the conceptual framework a corresponding coding pattern is introduced; a coding pattern is a guideline for the translation from UML to code of an element from the conceptual framework.

This approach supports a spiral course layout [5], reinforcing the most important concepts several times in the course. There are two criteria for the design of the spiral layout: the most common concepts of the conceptual framework are introduced first, and throughout the course the students must be able to create working programs.

The conceptual framework is comprehensive; for CS1 we restrict the coverage to association, composition and specialization which by far are the most used concepts in object oriented modeling and programming.

The starting point is a class and properties of that class. One of the properties of a class can be an association to another class; consequently the next topic is association. This correlates nicely to the fact that association (reference) is the most common structure between classes (objects). Composition is a special case of association; composition is taught in the next round of the spiral. The last structure to be thoroughly covered is specialization. Specialization is the least common structure in conceptual models, and it bridges nicely to the second half of the course where the focus is on software quality and design.

In the following subsections we describe some of the elements of the design of the course focusing on the first half of the course where modeling dominates.

3.1 Getting Started

We want to give the students an everyday understanding of object-orientation and a very informal understanding of the process of creating a UML class model. We therefore start by illustrating the concepts using everyday life situations in a role-play. The goal for the role-play is to illustrate structure and dynamics in terms of concepts, phenomena and messages in a problem domain and classes, objects and method calls in a corresponding (class and program) model. We use UML (primarily class diagrams) to describe concepts and their properties, without any formal introduction to the modeling language.

To introduce the students to basic coding we use a graphics package [6]. The graphics package is presented in terms of a class diagram; hence, the students experience very early the strength of a class model as an abstract description of a program component as well as a communication tool; the UML-model provides an effective "language" for documenting and communicating about classes.

This introductory part of the course provides an external view of classes and objects.

3.2 Class

After having *used* classes and objects we turn to an internal view and start writing classes; we do this by introducing the first coding pattern: *Implementation of a class*. The students discuss a domain concept, select a few properties, and express the domain concept using UML. Using the coding pattern the UML-description is systematically translated into Java code.

In this phase of the course the students learn about basic language constructs such as assignment, parameters, conditional statements; constructs needed for the systematic translation of model into code.

Initially, the coding pattern is introduced by example. Through a number of similar examples, the students become confident with the systematic translation, and finally they can generalize and create a generic coding pattern for a simple class.

3.3 Association

In the model of the problem domain the most common structure between classes is an association. We use several examples with progressive complexity to illustrate the concept and its implementation.

3.3.1 One Class with a Reference to Itself

Through a number of progressive examples we illustrate that an association is a property of a class, a class can have more than one association, and an association is a dynamic relation.

The students extend a previous example with a recursive association. One example is that a Person can be *married_to* another Person or the *lover* of another Person. This results in the model in figure 1.

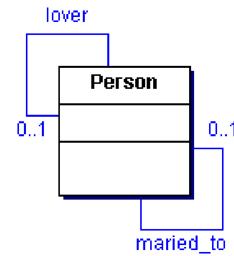

Figure 1: One class with two associations

In order to implement associations with 0..1 cardinality the student needs to know about programming language elements (e.g. reference and the null value). It also gives the students an understanding of interaction between objects (calling methods on other objects) and reference semantics.

Turning to 0..* associations imply that the student needs to know about Collections (either one of the Java standard Collections or the array type) and the need for iteration arises (an Iterator or an index variable and a simple loop). This is done using a simple algorithm pattern for sweeping through a collection.

3.3.2 More Classes

In order to get more interesting collaboration between classes, the next concept is associations between different classes. As a starting point we use a domain model with the following structure:

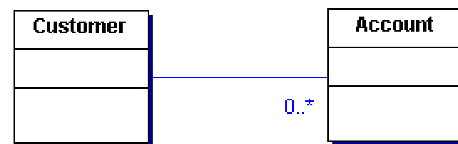

Figure 2: One customer can have many accounts

The students quickly understand that an association between different classes in principle is the same as a recursive association. This is true for the implementation as well; again the students generalize to a generic coding pattern for 0..* associations.

3.4 Composition, Specialization and Interfaces

We treat the remaining elements of the conceptual framework, composition and specialization, in a similar way. As mentioned earlier, specialization bridges nicely to the second half of the course focusing on software design and quality. The primary quality aspect is coupling and the main language construct by which to achieve low coupling is interfaces. Interfaces play an important role in the separation of specification and implementation: the specification of properties of a domain concept and (different) implementation(s) of these properties.

4. ON THE ROLE OF CONCEPTUAL MODELING IN CS1

In the following we will discuss some of the aspects of integrating conceptual modeling in an introductory programming course.

4.1 Systematic Approach to Programming

The goal is to teach the students to appreciate and achieve quality software. By good quality software we mean modifiable software, i.e. readable and understandable programs with a good structure, low coupling and high cohesion. These quality measures are by no means obvious to newcomers, and how to achieve them is even harder. We need to teach the students guidelines for achieving it and a vocabulary to talk abut their programs in order to help them build quality programs.

We reinforce five abstraction-levels of techniques for the systematic creation of object-oriented programs:

1. *Problem domain → model*: Create a UML class model of the problem domain, focusing on classes and structure between classes
2. *Model → Java code*: Create a skeleton for the program using the coding patterns.
3. *Functionality → Java code*: Specify properties and distribute responsibility among classes.
4. *Implementation of classes*: Create class invariants describing the internal constraints that have to be fulfilled before and after each method call.
5. *Implementation of methods*: Use algorithm patterns for the traditional algorithmic problems like searching, sweeping. Use loop-invariants for the systematic construction of loops.

In this paper focus is on the first two of these systematic techniques; we mention the other three to give the full picture of how we reinforce systematic techniques at different levels of abstraction.

4.2 Providing Confidence

To program is difficult! In [14] the authors found "shockingly low performance on simple programming problems, even among second-year, college-level students at four schools in three different countries". It requires knowledge and skills of many things such as the programming language, development tools and the capability of formulating a solution in such a way that a computer is able to understand it. Especially the last demand implies the need for creativity when programming.

Students find the creative process very difficult. In a more traditional programming course students are guided by standard algorithmic techniques such as searching, sorting, divide and conquer etc. The problem is that algorithmic techniques do not help the students to create the overall structure of a solution; they do not know where and how to start because the mental gap between the problem description and an implementation in terms of algorithms is too big. Conceptual modeling gives a systematic and structured approach to programming which provides confidence and a safe ground for addressing the programming task.

Most programming tasks are trivial and can be handled using simple standard techniques such as the generic coding patterns described above. By focusing on standard techniques first, the need for algorithmic creativity is reduced (and a thorough treatment is postponed to CS2).

4.3 The Programming Process

The modeling approach to programming invites for an iterative process where the program is developed incrementally. Through progressive exercises we reinforce such a process in order to imitate modern program development processes [4].

4.4 Abstraction

One of the important skills we want our students to possess is the capability to abstract. One way of stimulating the student's ability to abstract is to give several exercises with similar structure.

One example from the bank domain is the model shown in **Error! Reference source not found.**. In a student administration domain we have the following model:

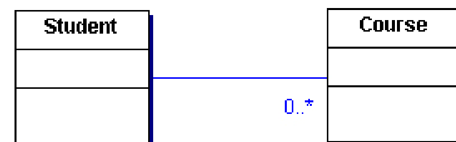

Figure 3: A student can participate in many courses

Initially the students see these two models as completely different, but gradually they realize they are both instantiations of the same abstract model:

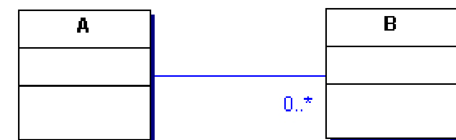

Figure 4: Abstract to many association

From this abstract model they can produce a corresponding generic coding pattern (see figure 5).

```
import java.util.*;

public class A
{  private Collection bs;
   public A()
   { bs = new ArrayList(); }
   public Collection getBs()
   { return bs; }
   public void addB(B b)
   { bs.add(b); }
   public void removeB(B b)
   { bs.remove(b); }
}
```

Figure 5: Generic coding pattern for 0..* association

5. CONCLUSIONS

In the recommendations of the Joint Task Force on Computing Curricula 2001 we have identified conceptual modeling as a lacking perspective in CS1. We have described the characteristics of conceptual modeling and argued that it is the defining characteristic of object-orientation. We have described a general structure for CS1 with conceptual modeling as the driving force. Furthermore we have discussed a number of aspects of this structure including a systematic approach to programming.

The approach described in this paper has been applied for three years. Apart from qualitative improvements as described above, the change has had a quantitative impact on the number of students passing the course. Before we started using conceptual modeling and other systematic approaches to the programming process the average percentage of students passing the exam were 52%. Since we switched to the approach described in this paper, the percentage of students passing the exam has increased to 79%.

6. FUTURE WORK

The approach to programming described in this paper was characterized by Kristen Nygaard as the Scandinavian approach to object-orientation. The COOL project (Comprehensive Object-Oriented Learning) is motivated by the following note on traditional ways of teaching object-orientation: "They suffer from the lack of a unifying perspective and a pedagogical approach focusing upon the modeling aspects of object-orientation" [8, paragraph 1.2]. Within the COOL project we intend to investigate the tension between pedagogical approaches, didactical techniques, suitable examples, tools etc.

7. REFERENCES

[1] Alphonce, C., and Ventura, P.J.: "Object-Orientation in CS1-CS2 by Design", *Proceedings of Innovation and Technology in Computer Science Education*, Aarhus, Denmark, 2002.

[2] Arnow, D., Dexter, S., and Weiss, G., *Introduction to Programming Using Java: An Object-Oriented Approach*, Addison-Wesley, 2004.

[3] Barnes, D.J., and Kölling, M. *Objects First with Java – A Practical Introduction using BlueJ*, Pearson Education, 2003.

[4] Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.

[5] Bergin, J., "14 Pedagogical Patterns", available on-line at "http://csis.pace.edu/~bergin/PedPat1.3.htm".

[6] Christensen, H.B., and Caspersen, M.E.: "Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS1", *Proceedings of the Fourth Australasian Computing Education Conference*, ACE 2000 Melbourne, Australia, 2000.

[7] Cooper, M. et al.: "Teaching Objects-First in Introductory Computer Science", *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, USA, 2003, pp. 191–195.

[8] Description of the COOL project, available on-line at "http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_COOL1.html".

[9] Fowler, M., *UML Distilled – A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2000.

[10] Horstmann, C.S., *Big Java*, John Wiley & Sons, 2001.

[11] Knudsen, J.L., and Madsen, O.L., *Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages*, DAIMI-PB 251, Department of Computer Science, University of Aarhus, Denmark, 1990.

[12] Lewis, J., and Loftus, W., *Java Software Solutions: Foundations of Program Design*, Addison-Wesley, 2003.

[13] Madsen, O.L., Møller-Petersen, B., and Nygaard, K., *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley/ACM Press, 1993.

[14] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. "A multinational, multiinstitutional study of assessment of programming skills of first-year CS students", *ACM SIGCSE Bulletin*, 33 (4), 2001, pp. 125–140.

[15] Niño J., and Hosch, F.A., An Introduction to Programming and Object-Oriented Design Using Java, John Wiley & Sons, 2001.

[16] The Joint Task Force on Computing Curricula (IEEE Computer Society and Association for Computing Machinery). Computing Curricula 2001 (final report), December 2001. Available on-line at "http://www.computer.org/education/cc2001/final".