Teaching Object-Oriented Programming

– Towards Teaching a Systematic Programming Process

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

Michael E. Caspersen
IT University West, University of Aarhus
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark
mec@it-vest.au.dk

Abstract

Teaching introductory object-oriented programming is considered difficult. We have developed a model-driven object-first approach with strong focus on systematic techniques and explicit exposure of the programming process. It is our experience that this is a very effective approach: the students learn object-oriented programming as well as fundamental software engineering techniques, and the dropout rate is down to a minimum.

Keywords: Object-orientation, objects-first, programming, conceptual modelling, model-driven, systematic programming, design by contract, programming process, apprenticeship, educational objectives.

1. Introduction

It is considered difficult by many to teach the basic concepts of object-orientation; this is evident from numerous papers and presentations at international conferences [16] as well as discussions on listservs [1].

One reason is that most textbooks –although they pretend differently– fundamentally have maintained a procedural view of programming [16].

Another reason is that most introductory programming texts have a syntax-driven organization of the material; the focus is primarily on the programming language (described in a bottom-up fashion starting with the simpler constructs of the language and then progressing to more advanced constructs). Only subordinate to the presentation of the language constructs follows the presentation of programming techniques. All too often programming techniques are not even explicit in textbooks.

A third reason is that many teachers, due to choice of programming language for the introductory programming course, are thrown into teaching object-orientation by accident so to speak and they really don't understand how to teach object-orientation [16]. In the last ten years many institutions have changed from Pascal to C++ or Java. While it is possible to switch to C++ and still teach procedural programming (C-style programming), this simply doesn't make sense for Java; Java is intrinsically an object-oriented programming language and therefore need to be taught that way.

It is a prevailing opinion that learning a programming language equals learning to program. In the call for papers for this workshop it is stated that "Switching to object-orientation is not just a matter of programming language". We suggest rephrasing and strengthening this statement: *Learning to program is not just a matter of learning a programming language*. In our opinion it is vital to embed the learning of programming in a context where

the primary focus is on learning systematic techniques to develop a program from a conceptual model of the problem domain and to train the students in the process of applying these techniques.

In this position paper we present our approach to teaching introductory object-oriented programming. In section 2 we discuss aspects of elementary learning theory that has played a central role for the way we have organized the learning process for our students. In section 3 we give an overall presentation of our current approach to teaching introductory object-oriented programming. In section 4 we stress the importance of explicitly teaching programming techniques, and we identify programming techniques at three different levels of programming. In section 5 we expand on the use of videos as a novel way of unfolding basic programming techniques to students, and in section 6 we wrap up and present our experience with the approach of which one important aspect is extremely low drop-out rates.

2. A Bit of Learning Theory

In this section we discuss aspect of elementary learning theory and how this theory has shaped the way we organize the learning process for our students.

2.1. Bloom's taxonomy and graduated exposure to complexity and structure

We have applied Bloom's taxonomy of educational objectives [8] as inspiration and guidance when organizing the learning process in an introductory programming course (for a programming-oriented exposition of Bloom's taxonomy see [13, pp. 14-15]).

Bloom's taxonomy consists of six categories ordered hierarchically:

1. *Knowledge*: The ability to reproduce material that has been learned. Levels: knowledge of facts, knowledge of ways and means to handle facts, knowledge of general principles and theories.
2. *Comprehension*: The ability to apply what has been learned but necessarily in way where it can be related to other material or thoroughly understood. Levels: translation (to one's own words), interpretation (summarize excerpts of the essential), extrapolation (implications and consequences).
3. *Application*: The ability to use general ideas, theories, principles, procedures and methods in specific (new) situations.
4. *Analysis*: The ability to decompose and uncover relations between individual parts. Levels: analysis of elements, analysis of relationships, analysis of organized principles.
5. *Synthesis*: The ability to reassemble the result of an analysis to a new whole. Levels: manufacturing of a new structure, generating a plan or planned operations, deduction of abstract relations.
6. *Evaluation*: The ability to evaluate a given material. Levels: evaluation by internal criteria, evaluation by external criteria.

Each category represents a level of understanding. The categories are cumulative i.e. each category depends on the previous categories; strictly speaking this means that category n is a prerequisite for category n+1.

Taken literally, the cumulative nature of Bloom's taxonomy indicates that the learning of a topic must be organized bottom-up in a strict fashion [13]. In [9] this is used as argument for a teaching method that has "[...] graduated exposure to complexity and structure based on levels of cognitive development."

Graduated exposure to complexity and structure can be achieved in many ways; complexity and structure of the programming language is the traditional approach, but we suggest another. In the previous section we brought forward the viewpoint that learning of programming must be embedded in a context where developing a program from a conceptual model is the primary focus. In order to apply the principle of graduated exposure to complexity and structure in this context, we have organized our course in such a way that the progression is dictated by complexity in the conceptual models that we use as starting points for all programming tasks; we start with very simple conceptual models and gradually add more and more structure and complexity throughout the course.

Contrary to general practice Kristen Nygaard [19] argued that the teaching of object-orientation should begin with sufficiently complex examples in order to expose the strength of analysing and describing complex situations in an object-oriented perspective. This analytic approach works well in isolation, but in the context of programming where the students have to implement the models in a programming language, it simply doesn't work; in this situation it is vital to begin with sufficiently simple examples.

We don't use a strict graduate exposure to complexity and structure though. A strict graduate exposure to complexity and structure is the analogue of waterfall methods in software engineering. Rather we use an iterative and incremental approach, a so-called spiral approach [6, 7].

A spiral approach requires an initial introduction to basic language constructs (the so-called big-bang problem identified by Pattis [20]). The problem can be approached in many ways; one of our solutions is to offer a simple context (a class library) with which the students are writing quite challenging programming within an hour [10].

In section 3 we discuss in more detail our organization of the introductory course in the light of Bloom's taxonomy and in particular the details of "graduated exposure to complexity and structure".

2.2. Apprenticeship

In the context of introductory programming, one important learning objective is the *processes of programming*. This means that it is regarded as important that the students gain insights into how programmers develop their solutions from the initial problems, e.g. how one frequently compile code, use documentation and test partial solutions. One way of attaining this goal is to expose the students to how an expert programmer works. The design of the introductory programming course must therefore include the learning of the process of creating solutions – not just the solutions itself.

This is described as a *decentred approach* in [21, 11]. Knowledge construction is considered as legitimate peripheral participation, i.e. the attention is on the student's participation in communities of practitioners where the old-timers (the teachers) legitimate the skills and knowledge of the individual newcomer (that is the student). The teacher therefore needs a much deeper understanding of the skills and knowledge of the students than traditionally can be obtained in the "lecture theatre" design of programming courses.

The apprenticeship approach is addressed further in section 4.

3. A Model-Driven Approach to Teaching Object-Oriented Programming

3.1. An integrated approach

In [15] 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 [12]: conceptual level, specification level and code/implementation level.

Most descriptions and discussions of the object-first strategy tend to focus on *instructing the computer* and *managing the program description*, and to our knowledge, no introductory textbook exists that includes conceptual modelling. We find it vital to balance the three views on the role of the programming language by including conceptual modelling in the learning process. 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.

The integration of conceptual modelling and coding provides structure, traceability, and a systematic approach to program development, and the integrated approach strongly motivate and support the students in their understanding and practice of the programming process.

A consequence of the integrated approach, and a remarkable evidence of its qualities, is that over a five year period drop out rates have gone down from 48% to 11%. For a further exposition of the model-driven approach, see [3, 4].

3.2. On conceptual modelling

In [17] 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 keyword 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.

Coding and conceptual modelling 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 of object-orientation, 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 [7], 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.

3.3. Hand-in-hand modelling and coding

In section 2.1 we discussed course organization in the light of Bloom's taxonomy and the notion of graduated exposure to complexity and structure. In this section we will briefly describe how we achieve this at the concrete level. We illustrate the hand-in-hand modelling and coding approach using a very simple example.

A typical early exercise is the following well-known example where we model (a very small but essential part of) the system for a bank. In the bank there are accounts and customers. Every account is owned by exactly one customer, and a customer can own any number of accounts. The situation is captured in the following UML class diagram:

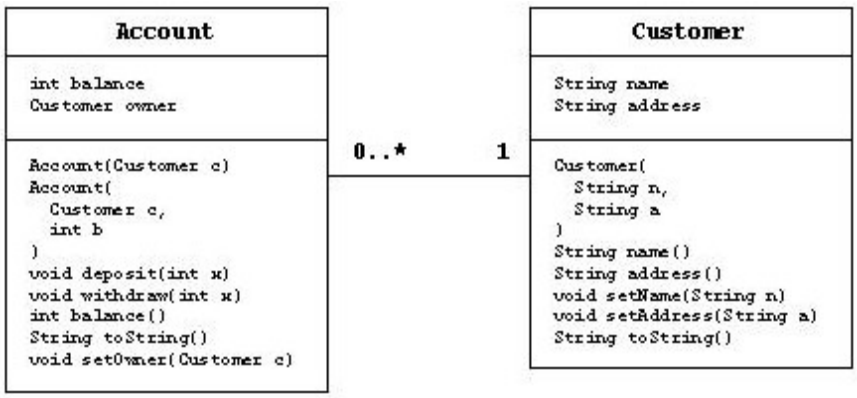

Figure 1: *UML class diagram for bank system*

The student will implement this model in a number of simple steps. Early in the course we guide the students via the text of the exercises and accompanying programming examples presented in class or in videos (see section 4) in which steps to take and in which order; later in the course we expect the students to be able to do this on their own.

In the specific example the students will start by ignoring the association and the methods to update the association. The students will start by implementing the constructor for the customer class; then they are asked to implement the inspector methods name and address, and then the mutator methods setName and setAddress. Then comes the constructor for the account class, then the inspector method balance, and then mutator methods deposit and

withdraw. Only after this we ask the students to deal with the association, first in one direction (the simple case) and finally in the direction from customer to account.

The point is that the program is developed incrementally, layer by layer so to speak. In general, this is a good way of developing software, but it is also a very efficient way of helping beginning students to separate concerns and deal with only one thing at a time.

Separation of concerns is perhaps the most important principle in the programmer's bag of tools; we return to a discussion of how we unfold this important principle in a number of concrete contexts in section 5.

4. Revealing the Programming Process

Revealing the programming process to beginning students is important, but traditional static teaching tools and materials such as textbooks, lecture notes, blackboards, slide presentations, etc. are insufficient for that purpose. They are useful for the presentation of the result of a process (a product), but not for the presentation of the process itself. Besides being insufficient for the presentation of a development process, the use of traditional tools and materials has another drawback: typically they are used for the presentation of an ideal solution which is the result of a non-linear development process. Like others [22, 23, 24], we consider this to be problematic; the presentation of the product independently of the development process will inevitably leave the students with the false impression that there is a linear and direct "royal road" from problem to solution. This is very far from the truth, but the problem for beginning students is that when they see their teacher present clean and simple solutions, they think they themselves should be able in a straightforward fashion to develop solutions in a similar way. When they realize they cannot, they blame themselves and feel incompetent. Consequently they will loose self confidence and in the worst case their motivation for learning to program.

It is important to create opportunities for the students to participate in an actual practice of programming experts so that they gradually learn through legitimate peripheral participation. This can be further operationalized by utilizing the different backgrounds of the students so they become each other's experts and legitimates in the shared learning community. Theoretically, individual knowledge is mediated by the apprentices' shared interests in learning object-oriented programming and by the ICTs and other resources (s)he has available. It is therefore important to create resources that unfold the programming process.

Besides teaching the students about the tools and techniques to be used for the development of programs (i.e. the programming language, programming techniques, IDE, etc.), we must also teach them about the development process, i.e. the task of using these tools and techniques to develop, in a systematic, incremental and typically non-linear way, a "good" solution for the problem at hand. An important part of this is to expound and demonstrate that many small steps are better than few large ones, that the result of every little step should be tested, that prior decisions may need to be undone and code refactored, that making errors is common also for experienced programmers, that compiler errors can be misleading/erroneous, that online documentation for class libraries provide valuable information, and that there, however non-linear, is a systematic way of developing a solution for the problem at hand. We cannot rely on the students to learn all of this by themselves, but using an apprenticeship approach we can show them how to do it; for this purpose we have used videos in the form of (screen captured) narrated programming sessions where the master shows how he creates solutions and by doing so unfolds the programming process.

For further exposition of this issue, and in particular on the use of videos to unfold the programming process, see [5]; see also [2].

5. Reinforcing Contracts and Systematic Programming Techniques

We identify contracts [18] and techniques for the systematic creation of object-oriented programs at four (six) different levels of abstraction:

1. *Problem domain* → ***conceptual model***: Create a UML class model of the problem domain, focusing on classes and structure between classes
2. *Problem domain* → ***Dynamic model***: Create a UML state chart to capture dynamic behaviour
3. *Conceptual model and dynamic model* → ***specification model***: Specify properties and distribute responsibility among classes.
4. *Specification model* → ***implementation***:
    a. *Specification model* → ***implementation of inter-class structure****:* Create a skeleton for the program using standard coding patterns for the different relations between classes.
    b. *Specification model* → ***implementation of intra-class structure****:* Create class invariants describing the internal constraints that have to be fulfilled before and after each method call.
    c. *Specification model* → ***implementation of methods****:* Use algorithm patterns for the traditional algorithmic problems e.g. sweeping, searching. Use loop-invariants for the systematic construction of loops.

In the introductory programming course focus is on the fourth level; beginning students cannot design [20], and therefore we provide a conceptual model/specification model as the basis of almost every programming assignment in the course.

We reinforce the notion of contracts at each level.

- At the conceptual level the contract is expressed as relations between classes; this contract is between the use and the programmer.
- At the specification level the contract is expressed as functional specifications of the interfaces (classes) in the model; this contract is between clients and implementations of interfaces.
- At the implementation level the contract is expressed as assertions in the program text (e.g. general assertions, class invariants, and loop invariants).

In the intro course we focus on contracts at the conceptual level and the implication of these contracts for the implementation in Java. It is our experience that the notion of contract in the context of a model-driven approach is a great help to beginning students.

6. Conclusion

We have presented our approach to teaching introductory object-oriented programming. The approach is characterized by being a model-driven, object-first approach with strong focus on systematic techniques and explicit focus on the programming process. It is our experience from the last five years that this is a very efficient approach: dropout rates have dropped from 48% to 11% in that period.

7. Acknowledgements

The current approach to teaching introductory programming has been developed and discussed be many people; in particular it is a pleasure to thank Henrik Bærbak Christensen, Ole Eriksen, Gudmund Frandsen, Annita Fjuk and Ola Berge for valuable discussions and

collaboration. We thank IT University West for financial support for the project, and we also thank the TA's and students participating in the courses for their patience, feedback and support.

8. References

[1] ACM Special Interest Group on Computer Science Education, Archives of sigcse-members@acm.org, http://listserv.acm.org/archives/wa.cgi?A1=ind0403d&L=sigcse-members#1; http://listserv.acm.org/archives/wa.cgi?A1=ind0403d&L=sigcse-members#2; http://listserv.acm.org/archives/wa.cgi?A1=ind0403d&L=sigcse-members#3.

[2] Astrachan, O. and Reed, D. "AAA and CS1: The Applied Apprenticeship Approach to CS1", *Proceedings of the twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, USA, 1995, pp. 1-5.

[3] Bennedsen J. and Caspersen, M. E. "A Model-First Approach to Teaching Object-Orientation", *Workshop on Learning and Teaching Object-Orientation – Scandinavian Perspectives*, Oslo, 20th October 2003.

[4] Bennedsen, J. and Caspersen, M. E. "Programming in Context – A Model-First Approach to CS1", *Proceedings of the thirty-fifth SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, pp. 477-481.

[5] Bennedsen, J. and Caspersen, M. E. "Revealing the Programming Process – Using Videos to Unfold Basic Programming Techniques", in preparation.

[6] Bergin, J. "14 Pedagogical Patterns", http://csis.pace.edu/~bergin/PedPat1.3.html.

[7] Bergin, J. "Pedagogical Pattern #32: Spiral", http://csis.pace.edu/~bergin/PedPat1.3.html#spiral.

[8] Bloom, B. S et al. *Taxonomy of Educational Objectives. The Classification of Educational Goals. Handbook 1: Cognitive Domain*, David McKay Company Inc., 1956. ISBN 0-679-30209-3.

[9] Buck D. and Stucki, D. J. "Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development", *Proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas, USA, 2000, pp. 75-79.

[10] Caspersen, M. E. and Christensen, H. B. "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, pp. 34-40. ACM Press, 2000. ISBN 1-58113-271-9

[11] Fjuk, A., Berge, O., Bennedsen, J. and Caspersen, M. E. "Learning Object-Orientation through ICT-mediated Apprenticeship", submitted for ICALT 2004, *the fourth IEEE International Conference on Advanced Learning Technologies*, Joensuu, Finland, 2004.

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

[13] Henriksen, P. *A Direct Interaction Tool for Software Engineering Education*, Master Thesis, Maersk McKinney Moller Institute for Production Technology, University of Southern Denmark, March 2004.

[14] Iyengar, S. & Soloway, E. (Eds.), *Empirical Studies of Programmers*, Ablex, New York, 1986.

[15] 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.

[16] Kölling, M. "The Curse of Hello World", *Workshop on Learning and Teaching Object-Orientation – Scandinavian Perspectives*, Oslo, 20th October 2003.

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

[18] Meyer, B. "Applying 'Design by Contract' ", *IEEE Computer*, Vol. 25 (10), October 1992, pp. 40-51.

[19] Nygaard, K. "A Sufficiently Complex Example", http://www.intermedia.uio.no/cool/complex.htm.

[20] Pattis, R. "The 'Procedures Early' Approach in CS 1: A Heresy", *Proceedings of the twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, pp. 122-126.

[21] Nielsen, K. and Kvale, S. "Current Issues of Apprenticeship". In *Nordisk Pedagogik*, Vol 17, pp. 130-139, 1997.

[22] Soloway, E., "Learning to Program = Learning to Construct Mechanisms and Explanations", *Communications of the ACM*, 29 (9), 1986, pp. 850-858.

[23] Spohrer, J. & Soloway, E., "Novice Mistakes: Are the Folk Wisdoms Correct?", *Communications of the ACM*, 29 (7), 1986, pp. 624-632.

[24] Spohrer, J. & Soloway, E., *Analyzing the High-Frequency Bugs in Novice Programs*, In [14].