Leave out the Modeling when Teaching Object-Orientation to Beginners

Axel Schmolitzky

Software Engineering Group
University of Hamburg, Germany
+49.40.42883 2302
schmolitzky@acm.org

Abstract. This position paper advocates an extreme position: Object-oriented Modeling (OOM) and Object-oriented Programming (OOP) should be kept clearly separate in teaching object-orientation in the first year, and OOP should be introduced first, with OOM following later. This position is based on several observations that were made in teaching OOP to beginners over the last six years. It is formulated deliberately as an extreme position to serve as basis for a lively discussion.

1 Introduction

Object-oriented programming (OOP) is an important building block in software engineering education. Students should be able to construct running code that serves its purpose. This requires several technical skills and a good understanding of programming language concepts.

Object-oriented modeling (OOM) is an important building block in software engineering education. Students should be able to identify and transfer originals in some application domain to object-oriented entities.

Thus students should become fluent in both OOP and OOM. The only question is: How to start? OOP first, modeling later? Modeling first, programming later? Or blend them, because OOP and OOM are tightly interrelated anyway? Another question is: When to start with OO? In the second or even third year, as an advanced topic, starting with imperative (e.g. with Pascal) or functional (e.g. with Scheme) programming in year one? Or should OO education start from day one, going objects first?

We have a clear position in this matter, grounded on our experience in teaching object-oriented programming and modeling to beginners over the last six years [ScZ07]. We believe that there should be a clear focus on OOP in the very beginning, leaving OOM to second semester earliest. We will try to explain our position.

The following assumptions are the venturing point of our discussion. While the thesis of this paper is quite extreme, these assumptions should not be too controversial for this workshop.

Java as the first programming language

Currently, the Java programming language is a popular choice for teaching OOP. Besides its economic advantages (Java has become a relevant language in commercial data processing), especially its comparatively simple object model constitutes an advantage in education: Java only knows one passing mechanism for parameters (per value) and all object types are reference types (manipulation of their instances is exclusively feasible via references); pointer arithmetic is impossible and the language itself guarantees that object boundaries are maintained. We have been using Java for several years in undergraduate and graduate education. Many textbooks exist for Java, as well as a large number of example programs.

Objects First

We apply an *Objects First* approach in teaching OOP, starting with objects from day one. BlueJ [KQPR03] is one tool that enables such an approach, and we have been using BlueJ for several years now in first year programming education.

The remainder of this paper implicitly assumes a teaching situation at a university where object-orientation is taught from day one and where Java is used as the first programming language.

2 The Differences between OOP and OOM

Before we can argue for a clear separation of OOP and OOM in teaching, we must obviously show that there are relevant differences between the two in the first place. This might seem difficult, because for an experienced software developer, programming actually *is* modeling. But the important word here is 'experienced'; with considerable experience in software development, programmers model in a problem domain by programming in it. They hardly recognize how much experience is necessary to intertwine these aspects smoothly. In the following we will try to dissect the relevant aspects of each.

2.1 What is OOP (and not OOM)?

A good metaphor for OOP is 'building machines.' Before we can build machines of some relevance, we should be rather adept at using the mechanics of their parts. The major parts for beginners are classes and methods, the minor parts are variables, statements and expressions.

A class (definition) in OOP is a blueprint for objects. It describes how objects hold their state (in *fields*) and how they operate on this state with their *methods*.

OOP is a reality of its own. There is a whole world of terms that relate to each other: primitive types, expressions, statements, blocks, variables, actual and formal parameters. While most of us (being teachers) have 20 years or more of practical and theoretical experience with these terms, beginners are easily overwhelmed by this abundance.

In OOP, the difference between compile-time and run-time is crucial; even in programming languages that are not being compiled a distinction must be made between static code and the dynamic execution of it. The metaphor of building machines fits perfectly here: We can distinguish the construction of a machine from its working in some environment. In OOM the difference between compile-time and run-time is irrelevant.

OOP can only be done using a formal syntax, because another formal machine has to understand the blueprints for our machines. OOM can be done without any formal syntax (even though UML can be used as a notation), e.g. using natural language (scenarios) or simple sketches on a whiteboard.

References are a central concept in OOP, especially with Java; without a solid understanding of references, almost no serious Java program can be written. In OOM (especially with UML), references are blurry, if existent at all.

Arrays are an important building block in OOP with Java, but they have no place in OOM.

2.2 What is OOM (and not OOP)?

Modeling means building *models*. A model is always a projection of some *original*. In OOM we need to know which parts of the application domain are important and thus should be included in the model and which parts are not important and should be left out. *This separation process is difficult*; as a matter of fact it is probably the most difficult part of software development.

A model always has a *purpose*. We build models to examine relevant properties of the originals. We build models to communicate about the originals, sometimes we build models to predict the behavior of the originals. Whatever we do with a model, it never exists *per se*, but for a purpose.

Anything can be an original for OOM. Organic entities are good examples of things that are relevant in OOM, but not in OOP. Consider real trees that grow outside our office windows (hopefully), with leaves and a need for water and sunshine. In some contexts, it makes perfect sense to *model* a tree, for example in an environmental simulation. But it is rather awkward to *program* a tree, at least for programming beginners. What are its operations? And what is its changeable state? Organic trees are not machines and as such no good examples for OOP.

In OOM, we model the *relevant attributes* of objects. An attribute is something that describes, colloquially speaking, a visible feature, e.g. a person has a hair color; a vehicle has a speed; an account has an account balance etc. In OOP such attributes must not necessarily be implemented directly as fields. Instead of having a field for an account balance, it is possible to record all account movements and calculate the account balance upon request. Vice versa, many fields exist that are not meant to be visible from 'the outside,' i.e. for clients of an object. For example, in an implementation of the data type *list* as a linked list, the reference for the list head is normally not a part of the interface. Using UML, this difference between field and attribute is typically blurred, because UML unifies these concepts under the term *attribute*.

In OOM there is no difference between the concepts *type* and *class*; in OOP a type is a more abstract notion (a set of elements in combination with the operations on these elements), while a class is an implementation.

2.3 Core terms in OOP and OOM

The following table gives a comparison of some core terms in both OOP and OOM, to further point out their differences.

The Concept...	in OOP is…	in OOM is…
Class	a blueprint for objects, a construction plan; always a static description of objects, sometimes an object as well	a result of classification; some concept common to several objects; a subsumption of generalizable properties
Object	an artifact at runtime, whose construction is based on the plan in some class code and whose state is held in its fields	any phenomenon in the real or an artificial world, organic or synthetic, that has a lifetime and a (potentially changeable) state
Operation	part of a type's definition; something we can do with the elements of a type's element set	an activity that can probe or change the state of some object
Inheritance	in a broad sense the subsumption of subtyping and subclassing; in essence an overrated mechanism for the incremental modification of code	used for hierarchies of classifications and all kinds of is-a-relationships
Name	something that variables must have, as well as classes and operations, but objects do not have names!	something we give objects (and classes and operations as well) to ease the modeling process.

In summary we can say that:

- OOP is more low-level than OOM (more mechanical, closer to the machine)
- OOP requires a model, while OOM requires an original.

3 The Question of Order

If we accept that OOP and OOM are different activities, we then have to ask: Should we begin with OOP or with OOM in education?

Several years ago we made an experiment: We started our course on OOP and OOM with the larger task of writing software for a taxi company. Students had no prior experience with imperative programming and were asked to start with a model of the application domain. Everybody knows what a taxi and a taxi company is, so we (naively) expected this to be an easy task. But the results were very disappointing, some even hair-raising. Because the students had no clearly defined concept of the 'target domain' of their modeling yet (objects in the OOP sense that model the objects of the domain), they modeled all kinds of things, up to the handset of the radio-unit used for communication between taxis and the taxi dispatch. Obviously we underestimated the complexity of the task for beginners.

The lesson we learned from this is: It is difficult, if not impossible, to model an application domain in some paradigm without at least some knowledge of the building blocks and their mechanics in the paradigm. Constructive modeling requires implicit knowledge about what is possible in the target domain. Students should at least have some experience in the domain of mechanical objects and their interactions before they can map originals to models.

We therefore largely provide the models in semester 1 and let the students implement these models (which is difficult enough for them) using OOP. Referring to the discussion of trees in section 2, we further completely avoid examples derived from nature (trees, mountains, animals and the like) and deliberately use *artifacts* as introductory examples in the first week of the first semester: an office chair with operations to change the seat's height, a TV with operations to change the channel, volume, color. Even a simple office table, although it might not offer any operations, is a better example than any organic one, because it is typically an instance of some blueprint that the manufacturer of the table has in its factory.

4 Conclusion

In this position paper we argued for a clear focus on OOP in the first semester of teaching object-oriented concepts and for introducing OOM earliest in the second semester. We pointed out the differences between OOP and OOM to give a clearer picture of what we mean by these terms.

5 Author's Biography

Axel Schmolitzky (also reachable under schmolitzky@informatik.uni-hamburg.de) has been a research assistant at the University of Hamburg, Germany, since 2001. He graduated in Computer Science at the University of Bremen, Germany, and holds a Ph.D. in Computer Science from the University of Ulm, Germany. He spent one year in the BlueJ-Group as a post-doc fellow of the DAAD (German Academics Exchange Service) at Monash University, Australia. From 2001 to 2003 he was also working half-time as a consultant and trainer for an Informatics Department spin-off company in Hamburg where he gained experience in industry projects and trainings. He has been actively engaged in restructuring the first year programming education at the Informatics Department. His main research interests are in the areas of object-oriented software construction, agile process models, programming language design, and educational issues in computer science.

References

[KQPR03] Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: "The BlueJ system and its pedagogy", *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13:4, S. 249-268, 2003.

[ScZ07] Schmolitzky, A., Züllighoven, H.: "Einführung in die Softwareentwicklung: Softwaretechnik trotz Objektorientierung?", in Zeller, A. and Deininger, M. (Eds.), *Software Engineering im Unterricht der Hochschulen (SEUH)*, dpunkt.verlag, 2007.