Teaching Object-Oriented Modeling and UML to Various Audiences

Sabine Moisan
Pulsar Team — INRIA
2004 route des Lucioles — BP 93
06902 Sophia Antipolis, France
Sabine.Moisan@sophia.inria.fr

Jean-Paul Rigault
Polytechnic Engineering School
University of Nice - Sophia Antipolis
930 route des Colles — BP 145
06903 Sophia Antipolis, France
jpr@polytech.unice.fr

ABSTRACT

This paper summarizes the experience of teaching object-oriented modeling and UML for more than a decade to various audiences (academic or corporate, software developers or not). We present the characteristics of the audiences. We investigate which modeling concepts are well accepted and which are not. We identify a number of general problems: background of attendees, object-oriented language as a prerequisite, limitations of tools, methodological issues, impact of company organizations. We propose (partial) solutions and a list of pragmatic advices.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Miscellaneous

General Terms

Software Modeling

1. INTRODUCTION

Both authors have more than 20 years experience in practicing and teaching object-oriented programming, mainly with C++, Java, and various object-oriented Lisp's, at beginner's as well as advanced levels. About 15 years ago, we started courses on object-oriented analysis and design, first using various object-oriented methods (Coad-Yourdon, Shlaer-Mellor, OMT) and we switched to UML [8] about 10 years ago. We have also run courses on Design Patterns [7, 5] and their application in C++ and Java, for more than 10 years.

We are teaching in academic as well as corporate contexts. At the Computer Science department of the Engineering School of the University of Nice, the students are preparing an Engineering degree in Software Development, roughly equivalent to a MSc. We also teach in-house short training sessions (a few days) for professionals belonging to companies dedicated to software development and applications. These companies are located mainly in France and the United Kingdom, but also in the United States and Australia.

Concerning object-oriented modeling, we have taught this topic to more than one thousand academic students and at least as many professionals.

This paper presents our personal experience and feeling with teaching object-oriented modeling to these audiences. These are not rigorously quantified experiments. The somewhat provocative style is meant to give rise to a debate. The paper is organized as follows. In the next section we present the various contexts of our teaching, the nature of the courses, and the characteristics of the audiences. In section 3 we remind the different ways of using UML and the impact on teaching. In section 4, we present our experience by listing what works well and what seems to fail. Then in section 5 we discuss some general issues. Before concluding, we summarize some pragmatic proposals.

2. CONTEXT OF THE TEACHING

2.1 Academic Training (MSc level)

The classes gather about 80–100 students for lectures, and about 20 for practical labs. Each course lasts usually for one semester (12–14 weeks) with about 3 hours per student and per week (1 hour of formal lecture, 2 hours of supervised practical labs).

As far as object-oriented modeling is concerned, we are presently running a UML semester course. The lab part consists of a case study which lasts for the whole semester and which is dealt with by four students working together. Most of the job is done as home work; the time needed is estimated as 3 to 5 hours per week and per student. Each week, the group of four students meets their assigned teaching assistant for about half an hour. During this meeting the group presents the work done during the week and discusses it with the assistant; then, the work for the next week is scheduled in common. No predefined methodology is followed, the students are just given a rough "work flow" describing the various tasks that they will have to perform along the project (see 5.5). The students are supposed to use tools, but no particular choice is imposed. Popular ones are Rational Rose or Rational System Modeler (which are freely available under the IBM academic initiative), or Poseidon, ArgoUML, Umbrello, **dia**, Bouml, various plugins under Eclipse, etc.

The audience is a typical student one: essentially technical background (knowledge of Java, possibly of C++, basic capabilities in algorithms and data structures), no experience of software development "in the large", little methodology awareness (just trial and error!), contentment with apparent result ("it works!"), no recognition of the importance of tests and maintenance. Some students have little perception of the difference between using a computer and developing software for it. They focus on short term programming, preferably in a playful way. Their motivation is not really for software development but rather for exam, mark, a line in their résumé. More and more often, they are lacking abstraction, analysis capabilities, and scientific culture, a general problem in our country(ies) where scientific studies attract fewer and fewer students.

2.2 Training for Software Developers

We also run in-house corporate training sessions for professional programmers. The companies are of various sizes (from a few to several thousands people). They are involved in software development, but with different maturity levels. They cover a variety of domains: data persistence, CAD in electronics, real time, networking and communications. They are mostly international companies, with employees from various countries, thus these sessions are usually taught in English. Each training session gathers from 6 to 12 attendees, lasts between 2 and 5 days, and is devoted to a specific topic. In the domain of object-oriented modeling our main offer to this population is a *UML for Analysis and Design* course but we also teach *Design Patterns in C++* or *in Java* (2 days), *Basic C++* (5 days), or *Advanced C++* (3 days).

The *UML for Analysis and Design* course lasts 3 or 4 days. It summarizes use case and scenario models using hierarchical use case diagrams, text scenarios in Cockburn's forms [2], and simple activity diagrams (this part is a digest of the *UML for Product Definition* described in the next section). Then it insists on analysis models using class diagrams, collaborations as a means to trace between use cases and their realizations, sequence diagrams as a formal way to express Cockburn's scenarios in term of object interactions, and state diagrams to describe the object behaviour (across all scenarios). Finally, the course skims over design models, introducing class diagram extended notations and refinements, state-charts (for "object protocols") and showing examples of component and deployment diagrams, as well as of code generation.

The training relies on a case study treated individually or by groups of two, which accounts for about half of the time. The objective is to sketch a use case and an analysis model for a simple application. The tool we use is usually Rational Rose, a frequent corporate choice.

The attendees are professional programmers of various ages: some are just getting out of school, others may have numerous years of practice. The first population is of course very similar to the aforementioned students, but with stronger and more realistic motivations since they are in "real life". The second population is indeed delicate to handle. These attendees often do not master object-oriented languages and they lack general culture in computer science. However, they have some culture in software development, positive by many aspects, but keeping them with old habits: for instance, practice of low level languages (C, even assembler) or of corporate environments that mask the difficulties and even the programming language itself (e.g., heavy use of macros in C and C++). They do not trust anything but code and may be reluctant to accept requirement models established by non developers! Their motivations are variable: some expect that the new acquired knowledge will help them to be promoted or assigned to more interesting projects; others come to the training because their manager order them to do so. Some will certainly use UML right after the training, others are just looking for general information.

Another difficulty is due to the multinational nature of these audiences and differences in background and culture. Some concepts or ways of teaching can be accepted by some people and rejected by others, depending on their country of origin. An example is the use of "role games", e.g., Class-Responsibility-Collaboration (CRC) card sessions that seem to be well accepted in the United States and that are rejected by many Europeans. If we were to dare an explanation, we would say that there is a tradition of "learning as a game" in the US. By contrast, in most European countries, learning is taken as a serious matter. Therefore, our students do not expect to "play" at school and they fear to expose themselves to a possible ridicule in front of their colleagues. This fear endures through professional life, but here the colleagues may also be their superiors. Moreover, it is not considered decent that a highly responsible (and highly paid) engineer play games when working! Thus, for students as for professionals, it seems to be both a problem of culture and of image.

2.3 Training for Non Developers

We also teach in-house corporate training sessions for non programmers. The session format is the same as previously (2 to 5 days, 6 to 12 attendees). In the domain of object-oriented modeling we offer an *Introduction to object-oriented and UML* (a sort of summary of basic concepts in 2 days) and *UML for Product Definition* (3 days).

The *UML for Product Definition* is the most interesting to discuss. It targets people working in marketing departments. They have to interact with the customers and to express customer needs in a way that developers should understand.

The course covers essentially use case and scenario modeling, with some insights in class modeling, sequence and activity diagrams, still remaining at "business level". More specifically, the course describes use case and scenario models using hierarchical use case diagrams, text scenarios in Cockburn's forms [2], and very basic activity diagrams. Then it skims over class diagrams, essentially using them for elaborating "conceptual models".

The training relies on a case study, processed individually or by groups of 2, and which accounts for half of the time. However, we usually do *not* use a UML modeling tool for this exercise, since teaching how to use the tool risks to take too much time and thus to mask the real problems.

The audience is, here again, rather heterogeneous. Some people have a pure marketing background, others may be

former developers. They all know their application up to a very detailed level. But they usually lack abstraction, scientific and computer culture. They are used to functional-like specifications and seldom accept the switch of perspective implied by use case modeling. They are suspicious of any change. For the other members who were formerly software developers, they form a delicate population because of their noxious desire to go into details of implementation! Finally, the motivations are variable, as for the previous course dedicated to developers.

3. WHAT USAGE OF UML TO TEACH?

Martin Fowler distinguishes three different modes of using UML [3]. In *UML as sketch*, UML is a communication tool among regular programmers and designers, used to describe and discuss on the fly about some delicate parts of the system to implement. The sessions are short and require lightweight tool support, possibly just a white board or a sheet of paper. The UML subset is often restricted to a few representations (use case, class, sequence, possible state diagrams), one at a time, and in their simplest form. According to Fowler, this mode is the most frequent, and is indeed the major choice in agile software development processes.

In *UML as blueprint*, the whole system (or a significant part of it) is thoroughly described. Completeness is the key issue, and precise documents are expected. In this mode, extensive tool support is required and modelers must be experienced designers since they do not model for themselves but for programmers to code up. Also a wider palette of UML representations has to be used, each corresponding to a particular point of view on the system, and these representations should be consistent.

In the third mode, *UML as programming language*, the idea is to build UML models precise enough to generate directly executable code. Thus the UML model becomes the source. Although many UML tools claim to be able to generate code, the result is still rather poor except for very restricted and stereotyped kinds of applications. The advent of Model Driven Engineering Architecture (MDA [4]) and its generalization, Model Driven Engineering (MDE [9]), raises legitimate hope. However, presently, *UML as programming language* is still a research topic, by many aspects. UML has not yet the syntax and, most importantly, the precise semantics to replace a modern object-oriented programming language.

Which of these modes should we teach? *UML as programming language* seems somewhat untimely, for corporate training. It is acceptable for Master students, combined with MDA/MDE in an elective course introducing them to research issues, at the end of their curriculum. For corporate training, the choice between the two other modes ideally depends on the company methodology and on the origin, background, and role of the attendees. Unfortunately, most of the time, we have to handle heterogeneous groups, which implies some trade-off. Teaching sketch mode is not so rewarding and attendees usually expect more from UML than sketches and drafts. Also, as it is normal for computer people, they expect to use tools. Unfortunately, most present tools are not very good at sketching! (See 5.4.)

So we usually adopt a "lightweight blueprint" approach, trying to achieve completeness[1], modeling with multiple views, and insisting on internal consistency of these views. If this approach is correctly assimilated, it is reasonable to think that students and professionals will certainly and naturally use the UML notation in sketch mode when communicating. This is confirmed by the experience as weel as by the global acceptance of the UML notation in sketch mode.

4. ACCEPTANCE OF UML

Acceptance of UML varies with the origin of the attendees. It is usually better for students than for professionals. Students do not have yet any habits in software development, the topic is new, the graphical nature of the models is rather appealing. The beginnings are sometimes difficult, though. The students are eager to code and modeling may initially appear as a waste of time. However, at the end of the semester and of their practical project, when they are asked about the usefulness of modeling, a vast majority answers that it was indeed useful and that this was a good way to clarify ideas before coding.

The situation is not as clear with professionals. For the most experienced, they are used to a way of doing things, which has often proved successful enough. Bringing UML into their practice shakes up their way of thinking, introduces an extra complexity without their anticipating any obvious benefit. Other professionals place too much hope in UML. They become disappointed when they realize that they got yet another language but not a panacea. In this case, they can reject UML as a whole in a rather harsh way.

In the following sub-sections, we examine some elements, either pedagogical or technical, that may impact acceptance or rejection, according to our experience.

4.1 The Importance of Case Studies

Everybody agrees on the importance of introducing a significant case study into UML teaching. Indeed, one does not teach UML as a programming language; in this matter the concepts are more important than the notation itself. To assimilate those concepts one must tackle case studies of significant size. In addition, some attendees also appreciate small exercises focused on particular tricky points of the course.

Choosing the topic of the case study is not obvious. Of course one should stick to "real life" problems and avoid too academic ones. For instance, we tried the following in our teaching: modeling a highway toll station, a city parking, a gas station, a command language for a (simulated) robot. These topics are rather data oriented (since most of our attendees are in this area) but we make sure that they contain significant state behavior. Moreover, for corporate training, it is our experience that the topic of the case study should be outside the professional concerns of the attendees, otherwise it is likely that the attendees know more about the topic than the teacher, drifting into endless debates and deep details, thus hiding conceptual matters.

It is an interesting question to know how to present the prob-

[1] at least within the time scale of a few days training session

lem statement of the case study. If it is slightly incomplete or even mildly inconsistent, professionals won't care because they are used to it! On the contrary, students will complain because they usually expect precise and complete assignment descriptions. We have to convince them that having a critical eye on problem statements and being ready to fix them is an integral part of the modeling job.

A last remark concerns team work. While we have ways to force students to constitute groups, it is much more delicate in corporate trainings. We have noticed that, especially in big companies, people are reluctant to work together when they do not know each other and there is no point in forcing them to group.

4.2 Well Accepted Modeling Concepts

Class and object diagrams

They are very natural for programmers and also for marketing people used to model data bases. The role of these diagrams as description and communication tools is easily understood. However, identifying objects and classes at the *correct* abstract level remains a difficult issue, even for people used to data base modeling. Indeed, if there can be "tricks", there are no absolute "recipes" to do it. Some advocate a sort of "linguistic approach", reading an informal specification or design document (depending on which phase you are in) and selecting nouns as candidates for classes and verbs as candidates for functions or methods. In our experience this does not work so well and nothing replaces a good understanding of the problem or its solution. For this, we simply promote group work, discussing and brainstorming in common.

Use cases and scenarios

They are considered as useful, yet bizarre, but tedious to elaborate. They induce two risks, though. The first is to restrict scenarios to diagrammatic representation, forgetting that scenarios are story telling [2]. Some people have problems in writing. Other are (too) proficient in this domain and can hardly respect the restricted style required by scenarios. The second risk with use cases is to go too far in the functional decomposition, entering into low level system details, forgetting about the external actors, and missing the point of object-orientation.

Activity diagrams

We use only the simplest form of activity diagrams to represent several scenarios, possibly a full use case. The similarity with traditional flow charts makes this representation easily understood. However, it presents a serious danger, identical to one of the previously mentioned use case risk: some attendees may continue the decomposition up to very fine grained activities, ending up with a functional approach and thus, here again, missing the point of object-oriented modeling.

4.3 Not So Well Accepted Modeling Concepts

State models

Apart from embedded system developers, most attendees have no conscience of the importance of state modeling and no practice thereof. State modeling, especially with state hierarchy, is hard. It relies on a rigorous mathematical theory and it may well be the part of UML with the most precise semantics. Moreover, many general tools are not very effective with these models, they do not generate code from them, nor allow simulation, creating frustration when trying to debug or validate the state models.

Sequence and collaboration diagrams

Universally recognized as precious in sketch mode but tedious and painful to do when it comes to blueprint! For traditional sequence diagrams, they are simple to use but their power of expression is limited, especially to express complex control flows. Thus many diagrams are needed to express the scenarios and their variants, often with common and thus repeated parts. The situation improves with UML 2 sequence diagrams, which are a more compact and modular representation owing to the notion of *fragments*. However, the syntax becomes complex and may require programming experience. It is sometimes so complex that a programming language (or pseudo-code) representation may be clearer and more natural. Moreover, many tools do not handle nicely fragment edition or update. Another problem about sequence diagrams is that some attendees wonder why using these diagrams since few tools generate code, simulation, or test from them.

Component diagrams

The component model in UML 1 is far from being clear and precise, and the gain of using diagrams for representing C/C++ .h or .c(pp) files is not obvious. In fact, here again, things are improving with UML 2 and the notion of composite diagrams, but at the cost of extra complexity.

Deployment diagrams

As defined in UML, they convey poor information. An interesting usage could be the mapping of component diagrams onto deployment ones, but this mapping is not supported by some tools (e.g., Rational Rose). For those who really need this sort of diagrams (e.g., to represent distributed systems), alternative models and tools are available, but they are not yet incorporated into UML.

Diagrams do not tell the whole story

Companion text, constraints (in free text form or in OCL), textual scenarios are at least as important, especially in requirement modeling. But elaborating diagrams is so time consuming and, on the other hand, illusorily seems to be such a rigorous activity that many forget this issue. Moreover, it is difficult to require a complete well written document within a short time scale, as in corporate training. It is much simpler for student assignments.

5. MAJOR PROBLEMS

Teaching software engineering is itself delicate. It is difficult to inculcate the importance of the different development phases as well as the culture of software product into the students. Compared with other engineering domains, software development has not yet reached a satisfying maturity.

In the sequel, we list a number of problems that all UML instructors have to face. We have no clear or universal solution for them and the debate is largely open.

5.1 What to Teach in UML?

UML is a big language, even bigger with UML 2. Moreover, the problem is not the language itself, but the concepts of

modeling. The arcane notations of UML 2 are not suitable for non-developers and for this population we usually restrict to classical (UML 1) class, sequence, and activity diagrams. Some UML 2 features can be useful for professional developers, provided they are supported by tools. Of course, academic teaching uses UML 2 notation, but only presenting major topics.

The Object Constraint Language (OCL) is not formally part of UML but related to it. It is neither simple not natural. More and more tools support it now and even are capable of generating code from the constraints. Introducing it to non software developers is almost impossible and the professional programmers usually do not like it: it looks like a programming language, but it is not; it has first order logic semantics, but it does not look like it. For our part, we do not include OCL in corporate trainings but we teach it in the academic curricula. However, we always insist on the need for constraints, if only in textual form.

It is important to know how to generate code from UML models. But it is even more important to have the student realize that (at this time at least) the process cannot be made entirely automatic. Many design or implementation decisions have to be taken that are not always represented in the UML model: for instance, attributes versus functions, template usage, refactoring for optimization, concurrency issues.

Last but not least, one should not forget that software modeling is not only elaborating models but also being able to read and use them. Thus, it is important to teach people to be also model readers. The obvious way is to show complete examples of models. A better way, in our opinion, is to have attendees complete unfinished models. A practical way that we apply in our case studies is to exchange models between groups to discuss and criticise the other's modeling choices.

5.2 Missing Background

Most attendees in our corporate sessions do not have a wide culture in computer science. Either they were initially trained in an other (hopefully scientific) discipline or their computer training took place a long time ago. There are several concepts of software that they do not master, in particular the notion of a *type* which plays a central role in the object-oriented approach, the difference between instantiation and specialization, between specialization and aggregation, or between a static mechanism and a dynamic one. However, these attendees have a good experience in software development for their domain.

In the academic world this latter experience is missing, but the theoretical concepts are usually more or less assimilated and the knowledge of programming paradigms is of course more recent—it is disappointing, though, to realize how quickly these concepts will vanish as soon as the students start working in the industry!

5.3 Object-Oriented Languages as a Prerequisite?

Coding is just one aspect of software development, and not the one which requires the largest effort. However, is it possible to compose a novel without knowing the language in which it will be written? And, is it not true that the language itself has some influence on the design of the work? After all, object-oriented methods and UML itself are born from object-oriented languages, not the contrary.

Of course, some aspects of UML do not require prior expertise in object-oriented programming to be understood and used: use case diagrams and simple sequence diagrams are of this kind. Beside, the knowledge of an object-oriented language does not help much in elaborating state diagrams, setting up efficient architectures, or even identifying objects and classes. However, learning an object-oriented language is certainly the best way to assimilate precisely the main concepts of class, method, generalization, polymorphism, software architecture, etc. A programming language is precise, rigorous, verifiable through a compiler, and liable to experiments.

On the other hand, too much emphasis on a particular object-oriented language leads to lower the abstraction level, to forget the targeted product, to ignore the semantic gap between specification and implementation. A solution, rather easy for academic students but problematic for corporate people, is to teach several languages of different kinds.

In our now long experience of corporate training, it has always been more rewarding and also more effective to teach UML to attendees with a prior experience of (object-oriented) programming than to ones without programming knowledge.

5.4 Problems with Tools

Computer science uses computer tools! It is even one of its main distinguishing characteristics among engineering disciplines. People like and request tools. However, present UML modeling tools are not satisfactory by many aspects, although the situation slowly improves with time.

Most general UML tools are not so easy to master, even for doing simple diagrams. They are immature and late with respect to the standards, or even worse, they have their own interpretation of the standards. Their ergonomics may be dubious. The model verification facilities are often poor, internal consistency of models is neglected, few simulation/execution tools are available, quality assessment means and metrics are lacking. Many tools are still closed. They ignore external languages (like OCL). The exchange of models through XMI is at best an adventure, at worst a joke. Some tools try to support "intelligent" guidance. The result may be more irritating than useful.

A particularly annoying issue is that of model consistency. UML proposes a wealth of different views on the system, at different levels of abstraction, corresponding to different phases of development. This is good! But these views are not independent, they have strong and consistent relationships. Most of these relations are ignored by tools and cannot be introduced by the modeler. As a matter of example, virtually all tools handle correctly a relation between sequence diagrams and class definitions. When drawing a message in the sequence diagram, one can conjure up a menu with the list of all operations of the receiving object. But when drawing a state machine representing the "protocol" of

a particular class, where most of the transitions are labeled with one of the class operations, most tools, if any, do not show the same magic menu.

More generally, one of the major critics that many tools incur is that they are blueprint oriented and not easily usable for exploratory modeling and sketching. When you have already a good idea of the various aspects of your UML model, possibly because you have drafted it on paper or white board, the tools help you to enter it efficiently. But they make it difficult to modify, refactor, or even update it afterwards, especially when the changes impact interrelations between views. Try to simply permute two objects in a sequence diagram with fragments. Try to add an operation in a class and introduce this operation as target in some of your sequence diagrams. And how many times did you have to reorganize your diagram layouts after a slight modification? An other significant example is the support of best practices such as Design Patterns. Several tools allow you to set up a particular pattern by selecting classes and operations in you model and assigning them a role in the pattern. This is really nice. But what happens if, afterwards you change your mind and wish to modify, even cosmetically, the pattern?

Many tools advertise generating source code (usually from class diagrams only), which is often out of place. It is an illusion to believe that drawing simple diagrams will be enough and that code will automatically follow. The frustration is great when one realizes that the effort to make diagrams generate correct code is at least as important as writing the code itself.

5.5 Which Methodology?

Indeed UML is independent from methodologies. However, can UML be *taught* independently of any methodology? A well defined software development process should define the elements of UML (views and diagrams) to be used, the order of their elaboration, and most of the needed relationships between them. These questions are absolutely recurrent among the training attendees, especially—but not only—among the students, although this issue depends on the attendee origins.

For academic training it is difficult to chose one specific methodology: there is no consensual choice; the differences between methodologies, even superficial, may disturb students; it would be somewhat difficult for the teaching staff to come to an agreement on a given methodology. Moreover, in the industry, most methodologies are defined inhouse and are likely to be much different from the one used for teaching. Finally, on a pedagogic point of view, it might not be a good idea to impose too strict a methodology on students. The risk is that they apply it blindly, without getting a chance to discover by themselves the ins and outs of methodological issues. To be honest, students usually do not appreciate this idea! A possibility would be to teach a general methodological framework such as the Rational Unified Process (RUP [6]) but it requires customization to be used, which raises the aforementioned problems. In our case, we decided to use a ultra-lightweight methodology for student assignments. We define a simple flow which describes a partial order for elaborating views and diagrams, insisting on consistency relations among views.

On the other hand, professionals already follow a more or less defined process. They are often affective about it: they like it or they hate it. Those who like it wish to know how a UML approach would integrate smoothly into their present way of working and what will have to change (something they are highly reluctant to do). The others expect to discover how UML (considered not only as a notation but, misunderstandingly, as a full process) could be a replacement for their dreaded in-house process. Answering these questions requires that the trainers have a good knowledge of the company culture, habits, and processes, which is seldom the case.

5.6 Evaluation of the quality of models

"How do I know that my model is good?" is a recurrent question, from all attendees, students or not. It is also an embarrassing question. Setting up formal measurements and metrics on models is yet a research issue. Of course, common sense and style considerations can play a role. The models should have a clear organization, they should go directly to the point, they should not introduce unncessary artifacts, inter-relations between views should be kept to what is needed, no more. Should we teach modeling like some artistic disciplines are taught (writing classes, master classes, architecture)? Indeed there are demands for elegance and even beauty in software architecture that software developers and students are usually not conscious of. Unfortunately, these criteria are not objective and their appreciation requires a good experience in modeling.

For our part, we insist on the robustness of models. We play a "what if" game with the attendees to evaluate the impact of changes or evolutions on their models. More perniciously, we also change the problem specification near the end of the case study time (we do this only with students!).

5.7 Impact of Company Organization

The organization and management of a company, the way its development process is defined and enforced, the individual implication of its employees have an impact on the introduction of object-oriented modeling. Of course this impact is only relevant for corporate training.

Notion of a "seamless process"

Although promoted by several authors, seamlessness is unrealistic for most companies that we know, especially the biggest ones, where there are multiple jobs, several skills, different guilds. Beyond technical skills there are also political and organizational reasons that make it impossible to have the same people in charge from specification to final tests and maintenance. This separation of roles is paradoxical with the "seamless process" proposed by UML. Indeed, this seamlessness often blurs the limits of abstraction levels and allows people to mind about topics which are none of their business. For instance, we already mentioned the tendency of some former developers to tweak requirement models toward implementation. Only a clear and enforced methodology could prevent such drift.

Problems with management

In companies, it is not sufficient to train base developers

if the middle and higher management do not commit to the object-oriented modeling approach. Thus the managers should be trained too! This is the reason why we propose a one day session of *UML for Executives*. Unfortunately, few managers have time or desire for such a training, which is a pity because they are the ones who decide about integrating object-oriented modeling into corporate methodology.

In-house methodology
It is difficult to run corporate trainings without knowing the context, the habits, the methods used in the organization. Not all companies are willing to expose their development process because of corporate secrets or just technical modesty. Besides, it is often the case that the same methodology is not used in the different departments or teams of a training session attendees. On the other hand, external trainers cannot master a company methodology and it is not even desirable. When hiring an external training provider, most companies, and the attendees themselves, expect some view from the outside, not yet another in-house vision. The tradeoff is not easy to find. In our case, we try to remain neutral and to stick to our lightweight process (see 5.5).

Another difficulty is when (object-oriented) modeling is not part of the enterprise culture and when a smmoth integration of UML into the development process has not been anticipated. In this case, UML is perceived as yet an other tool, just like C++ or J2EE, complex to grasp, and problematic to integrate into everyday practice.

6. PROPOSALS

In this section, we summarize some suggestions, stemming from our experience, to conduct trainings in academic or industrial context. All these suggestions have been discussed in the previous sections.

All trainings
The following advices apply to all sorts of UML trainings, independently of the population and the duration.

- Insist on the role of modeling as a step in problem solving and a way to clarify informal specifications and to formalize design.
- Try to remain methodology neutral whenever possible. Just provide general guidelines to schedule modeling activities.
- Do not try to teach all UML(2). Select concepts and views relevant to the target population.
- Insist on a model not being just a set of diagrams. Constraints and textual information are mandatory.
- Run a case study in sync with formal lectures. The topic should be realistic and fit within the time frame.
- Teach to read models, not only to write them.
- Use computer tools but do not expect too much.
- Improve models by asking "what if?", even by changing specifications.

Academic training
As faculty members, we (sort of) master the contents of our courses and curricula and, hence, we know the background of our students.

- Make sure that technical prerequisites are met: practice of at least one object-oriented language, familiarity with computer science concepts such as types, state-machines, inheritance, aggregation, polymorphism.
- Emphasize the links between modeling concepts and these theoretical aspects whenever possible.
- Force students to work in team (a group of 4 seems a good choice) for the case study.
- Do not spoon-feed students by providing a clear-cut problem statement for the case study.
- Give students a free hand on methodology as long as they seem to follow one.
- Do not compromise on deadlines, quality of documents and presentations.

Corporate training
Here we are just service providers. We do not decide how corporations behave and we cannot master their culture, their mentality, their background, or even their choice of tools and methodology. Even if we define prerequisites for our training sessions, they may not be fully respected.

- Try to understand why the company asks for UML training.
  - What are the kind(s) of methodology used within the company?
  - How is UML integrated or integrable into these methodologies?
  - How does management feel about object-oriented modeling?
- However do not try to stick to the company methodology, if you do not master it.
- Remember that the case study lasts only for a few days.
- Choose a case study topic in the attendees' general area of interest but not in a domain where they are specialists.

7. CONCLUSION

Due to the practical and operational nature of software engineering, in object-oriented modeling more than in many others, it is worthwhile to run training both in the academic world and in the industrial one.

This paper analyzes some problems that we met in running courses in both context and lessons that we have drawn. It also proposes some simple advices. We believe that more formal studies are needed, although it might be problematic in corporate trainings.

In the academic context, things are rather simple. We master students background, we can relate empirical modeling and theory, we have time, we control the flow of the case study. This is not true in short term corporate sessions. For these latter, the key to the success lies in understanding the motivation for organizing UML courses. Discussing with head executives in charge of sofware development may be beneficial. Not all companies are aware that it is useless to introduce software engineering methods and tools into an organization which has not reached the proper maturity level, as pointed out by the CMMI [1].

When a company fails to integrate UML in its software methodology—and we have already mentioned several reasons for which it may occur—one may hear that "UML is the problem, not the solution". There are no silver bullet. Our job as trainers is to ease the integration of object-oriented modeling, to avoid UML being a problem, but also to have people realize that UML by itself cannot be *the* solution.

8. REFERENCES

[1] Carnegie Mellon Software Engineering Institute. Capability maturity model integration. http://www.sei.cmu.edu/cmmi/, 2006.

[2] A. Cockburn. *Writing Effective Use Case*. Addison Wesley, 2001.

[3] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, 3rd edition, 2003.

[4] D. S. Frankel. *Model Driven Architecture*. OMG Press, Wiley, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[6] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley, second edition, 2000.

[7] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall Ptr, 3rd edition, 2004.

[8] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 2nd edition, 2004.

[9] T. Stahl and M. Völter. *Model-Driven Software Development*. Wiley, 2005.