Teaching Object-orientation: From Semi-colons to Frameworks

C Johansson
EP Frameworks/University of Karlskrona/Ronneby

L Ohlsson & P Molin

University of Karlskrona/Ronneby, Soft Center,

S-372 25 Ronneby, Sweden

Abstract

Software engineering is often associated with the software technology of yesterday. To manage anything, predictability is a key issue, and it takes some time to gather the necessary experience of a new technology. Object technology has not yet reached this stage of maturity but is rapidly becoming established in industry. This paper describes how the object-oriented paradigm is used throughout an academic software engineering education program. The relative immaturity of existing methods and practices is handled by an experiential learning approach that emphasizes questions rather than answers. The paper discusses various pedagogical aspects as well as some of the practical trade-offs and experiences gained in the implementation of the program.

1 Introduction

The purpose of a software engineering education program is to promote knowledge and skills which consolidate state-of the-art software technology with effective methods for managing and controlling the software development process. Another way to express this goal is the development of a professional attitude to software development. To teach professionalism in an academic environment, the three year software engineering program at the University of Karlskrona/Ronneby relies on the principles of role playing, learning by doing and commitment culture, Ohlsson[1]. Where applicable, these principles are used in all courses in the program, but primarily in the series of project courses in which teachers participate as customer representatives, project managers, heads of department, and quality managers. The illusion of a real-world development situation is further enhanced by using commercial products as models for the applications built in the projects as well as employing temporary teaching staff from some of the 50 software companies located on campus. The procedural knowledge characteristic of software engineering also requires a tighter integration of the different courses than in many other academic programs.

As the technological foundation for this education in professionalism, we have chosen to concentrate on object-oriented technology. Object-orientation forms the core of several courses in the program and has a strong influence on a number of others. This paper describes how we have designed these courses and how we have managed to integrate the material covered with the aim of teaching professionalism. The courses described in this paper are:

- Object-oriented Programming in C++
- Data structures and Algorithms in C++
- Object-oriented System Development
- Individual Software Engineering Project
- Database Design
- Small Team Software Engineering Project
- Large Software Engineering Project

A key question in any software course is which programming language to use. Factors to be taken into account are both pedagogical, such as the kind of mental image the language encourages; and practical, such as the tools the students may use for the language on the installed hardware platform and within the budget allowed. Typically, these concerns are in conflict, and the best solution must therefore be a compromise. We have chosen C++ as the main programming language throughout. The main benefit of C++ is that, due to its increasingly widespread use in industry, it is supported by a large spectrum of development tools. Its compatibility with C allows us to choose other systems used in the program, for example, database management systems, basically without restriction. Using the same language in most courses minimizes distraction caused by language details in courses with no programming focus. The drawbacks of C++ are, of course its awkward syntax and the fact that it only supports, but not enforces object-oriented thinking. In our experience, however, these drawbacks may be overcome by suitably designed courses. In fact, we like to think of them as opportunities to learn other, generally useful principles: awkward syntax can be handled by a disciplined coding style, and the lack of enforcement of a particular way of thinking illustrates separation of mechanisms from policy.

2 Object-oriented Programming in C++

From the beginning of this course, syntax issues are de-emphasized in formal teaching. When program fragments are presented, the emphasis is on what they express rather than how they do it. We try to rely as far as possible on imitation when learning language details. Basic concepts such as datatypes, simple statements and loop constructs are quickly covered in just a few lectures. Although we wish to introduce classes early, we have learned from experience that it is better to start with functions when presenting the idea of abstractions. Functions as black boxes show how something may be either a primitive or complex entity depending on one's point of view.

The first class, which is introduced during the second week, is a simple account class with deposit and withdrawal methods. It is shown how classes extend the notion of functional abstraction. The difference and relation between specification and implementation is given considerable attention. Dynamic pointer types are treated in detail, in particular in inheritance hierarchies in conjunction with dynamic binding, and the important difference and relationship between inheritance and dynamic binding for reaching upwards and downwards respectively in the class hierarchy. Along with dynamic binding comes the distinction between abstract and concrete classes. In our experience, it was difficult to find the right order to teach these issues. Our current order, which seems to be functioning well, is as follows: We start with simple public inheritance without private and protected variables, and then successively introduce protected variables, access methods, static binding with automatic variables, abstract classes and virtual functions, and finally, dynamic binding with dynamic variables.

As this course is part of a software engineering program, we attach importance from the beginning to coding standards and conventions. All the examples shown during lectures follow the design and programming guidelines that will be used throughout all courses in the program, though the guidelines are not formally introduced in this course. The guidelines cover naming of classes, variables and functions as well as how and when to comment. Descriptive variable names, proper inheritance hierarchies, proper layout and indentation, etc. are required to pass the laboratory sessions and the examination.

The laboratory series consists of five sessions. Each laboratory assignment in this course as well as in the other courses may take up to several days to complete, but teacher assistance is limited to four hours per exercise. The first session is about C++ functions of approximately 10-15 lines of code, and the second session covers pointers. The third session introduces code reuse by presenting a program consisting of a few classes which should be modified. In the fourth session the students write a small text editor and the fifth session presents the well-known shape class to provide practice in dynamic binding.

3 Data structures and Algorithms in C++

Data structures and Algorithms in C++ are the successor of Object-oriented Programming in C++. In this course we use the free-ware library LEDA, (Library of Efficient Datatypes and Algorithms) Näher[2], which contains source code for classes that implement simple data types such as strings, vectors and matrices, and basic data structures such as stacks, queues, lists, sets and graphs. The students use LEDA in two out of four laboratory exercises: The vector library for implementing a matrix class, and the string and list class for implementing a glossary. The implementation of LEDA is optimized with respect to performance, and thus makes use of several tricky constructs. This gives the students an opportunity to learn to read more complex code than usually found in conventional text books and also encourages understanding of how to write efficient C++ programs.

This course provides a natural context for introducing parameterized classes. The text book used, Budd[3], describes all the classical data structures as C++ templates; and since we have now upgraded to the latest version LEDA, templates are used in the laboratory exercises as well.

We use an established set of programming rules and recommendations (an enhanced version of Ericsson's official guide Ellemtel[4]) for all the examples in the courses, but this guide is not formally introduced, however, until the end of the course. This has proven to be the stage when the students have reached the appropriate level of maturity and can appreciate the benefit of a defined style guide.

4 Object-oriented System Development

We believe that it is important to cover object modeling early in the program, and we have even played with the idea of teaching object-oriented analysis before the first programming course since it is, in a sense, independent of programming languages. But without a basic knowledge of how it will be used, the students lack motivation for something they perceive as mainly philosophical. This course is therefore given after the two C++ courses described above.

The choice of method for this course poses a dilemma similar to that posed by the choice of programming language. We have found OMT (Object Modeling Technique), Rumbaugh[5], to be suitable as a first method. It uses an easy-to-follow step-wise modeling process, has simple notation, and is supported by a well-written text book.

For the practical modeling sessions, we use a group dynamic approach. Groups of about fifteen students work on the same model, using white boards and, if necessary, plastic sheets on the wall. One or two students act as model secretaries while the group is brainstorming ideas, expressing opinions and making suggestions to improve the model. Discussions may deal with such questions as "Is this really a class?" or "Shouldn't that class be a sub-class of that one?". The student's eager participation in these discussions often makes the teacher's role superfluous.

As an exercise in making a complete object model, the students use the OMT-tool to develop both an object model and a dynamic model for an exchange machine for different currencies.

A topic recently included in this course is that of patterns, Gamma[6], as we anticipate that pattern handbooks will soon become a standard tool for the standard software designer. Patterns express guidelines, standard techniques and useful practical tips about how to achieve a good design. Patterns are a way to express not only the what and how of a design, but also why something is designed as it is. We currently teach patterns by letting students search places in existing code where some already documented patterns have been applied or even in cases where such patterns should have been applied but have not been so.

Being confined to one method gives the advantage that little time and effort need be spent on notational details which can instead be spent on deeper issues. The drawback is that it gives exposure to only one particular view of object modeling. We think it is important to have alternative approaches in order to be able to form a proper personal opinion on the matter. Included in the course is therefore the study of the literature related to one alternative method, chosen from the list of Booch, OOSE, Coad/Yourdon, SBM, BON, Wirfs-Brock, Embley et al and Firesmith. A written report is handed in which compares and contrasts this method with OMT.

Included in this course is also an introduction to basic software engineering techniques such as requirements specification, project planning and software testing. These topics are only touched upon in this course as a priming for more elaborate coverage in subsequent courses.

5 Individual Software Engineering Project

This is the first project course and it is run as a role-playing game with teachers as customers, and students as contractors. The applications developed may be a game or a simple utility application such as a register program or a mail tool. The emphasis of the course is on negotiating and managing a contract for the project. Typically the task is given by pointing to some existing program and asking for a variant of this. The first thing the students do is write a specification of the product and gain the agreement of the client as well as decide on a method to demonstrate progress made.

Very little guidance is given on how to organize the project. Our aim is to allow them to make mistakes. For example, the students often spend too little time on analysis and design. And, as is usual in projects involving immature students, the rules and guidelines for good design and coding are often abandoned under the pressure of a deadline. Towards the end of the course, students usually recognize their mistakes, thereby gaining motivation for the more advanced techniques introduced in subsequent courses.

6 Database design

The database course consists of two parts: relational database design and object-oriented database design. It builds on the analysis part of the object-oriented system development course. The students design a rather large model for each of the two kinds of databases. Included in the part on relational databases is the transformation of an object-oriented model to a relational model, including, for example, the translation of inheritance structures to relational tables.

The object-oriented database part extends object-oriented modeling by including database concepts of persistent storage, transaction handling, indexing, etc. The database system used is ObjectStore, which uses an extension of C++ as the data definition and manipulation language.

This course also covers user interface design since there is no course in the program specifically dedicated to the subject. The last laboratory session consists of the construction of a window-based graphical user interface using object-oriented user interface building tools.

7 Small Team Software Engineering Project

The main point of this course is to introduce the difficulties involved in working as a group to develop software. In teams of four to five people the students practice dividing the development work, not only into different phases, but also between different individuals. Other points of emphasis are project planning, progress tracking, and configuration management. Fundamental to planning and tracking is the ability to make estimations about time, resources, size and complexity, although no formal methods are presented in this course. Experimenting with metrics such as number of classes, number of methods, number of lines of code, complexity, and other object-oriented related measures are used to gain experience of answering questions such as "How long are the different phases in an object-oriented project?"; "How long time does it take to build x classes?"; "How long does it take to make changes to x classes in y hierarchies?" etc. The use of the previously introduced programming rules and recommendations becomes essential here, otherwise there is no basis for building up reliable experience from these estimation experiments.

In this course the teams have the freedom to use an object-oriented language other than C++, such as Smalltalk or Eiffel. To learn these languages, the students must rely on text books and manuals only, because there is no formal teaching of these languages. The first time we allowed this freedom of language it happened almost by accident, and it surprised us how quickly the students became productive with a completely new language.

The course consists of two parts: one where a new system is built, and one where changes and extensions are made to a result from the first part. In order to provide some realism to the maintenance project the teams switch systems with each other after the first part. Not surprisingly, animated discussions between the teams on what constitutes good object-oriented design frequently occur during the maintenance project. The knowledge that their work will be scrutinized by others also makes the students highly motivated to follow the programming rules and guidelines introduced in the previous courses.

8 Large Software Engineering Project

This course consists of developing Object-oriented software in teams of approximately twelve to fifteen people. The total amount of student-power resources available for this course is some 50-60 man-months. Issues emphasized in this course include quality management, work in pre-project phases, and project bureaucracy with a well-defined process model documentation standard.

The system developed is often some kind of domain-specific object-oriented framework. An example is the system developed last year which was a general gateway for receiving, reformatting, archiving, logging and forwarding of so-called customer service orders to different network elements in a mobile telephone network. A network element could, for example, be an exchange, and a service order could be adding subscribers to an exchange or receiving billing information for a particular subscriber. The gateways which are in use today are tightly coupled to specific communication protocols and particular network elements from relevant vendors. The general gateway framework application was instantiated for one specific vendor, but the framework itself was made completely independent of the kind of network element, the format of service orders, the command languages on session level and higher OSI-levels, network protocol, and user interface.

ISO 9000-3 is used as the basis for the quality activities in the project. Reviews and inspections are formalized with written rules and meetings for which minutes are recorded. The organization is clearly defined, document rules are developed, roles and responsibilities are defined, change control is formalized, etc. And these aspects are all controlled through project process audits.

Software metrics are further practised in this course, although we still do not know which aspects of an object-oriented system are important to measure. We believe, however, that valuable insights are gained by experimenting.

9 Conclusions and reflections

There are some practical educational benefits from the software engineering techniques taught in this program. One worth mentioning is the widespread use of reviews and inspections. In the early courses, the teachers have the traditional role of giving comments and feedback on how well the various assignments have been carried out. But in the project courses this role is naturally taken over by peer groups, which makes the students continue to improve their skills in object-oriented techniques without any direct teacher effort.

We have found it important to have the first project course early on in the program. It is often during the individual software engineering project that the students first experience being in control of the software medium. This feeling is, of course, then alternatively shattered and regained several times during the program as further levels of complexity are successively introduced.

It is preferable to run the courses at half-speed, i. e. a five week course spread out over ten weeks. This gives a balance between the need for new concepts to mature over a long time while at the same time avoiding giving the students too many subjects to keep in focus at the same time.

The program is currently being extended to a master's degree in software engineering. Among the topics to be covered in more detail are specification and testing methods for object-oriented software.

Feedback from employers of students who have graduated from the program has been very positive. It seems the students quickly become involved in real projects, working as designers, programmers, documenters and participants in inspections.

The emphasis on experiential learning has worked surprisingly well. The students learn as much from discussions with each other as they do from what they are taught by the teachers. As a result, the level of creativity and motivation is significantly above average. And contrary to common belief, the large amount of practical work gives the students a better understanding of theoretical aspects.

10 References

[1] Ohlsson L., Johansson C., An Attempt to Teach Professionalism in Engineering Education (ed. T. V. Duggan) pp. 319- 324, Proceedings of the 3rd World Conference on Engineering Education, Vol 2., Portsmouth, 1992.

[2] Näher S. , LEDA User Manual, Max-Planck-Institut for Informatik Im Stadtwald, Saarbrucken, 1993.

[3] Budd T. A., *Classic Data Structures in C++*, Addison-Wesley 1993.

[4] Ellemtel Telecommunication Systems Laboratories, Programming in C++, Rules and Recommendations, 1992.

[5] Rumbaugh J. et al., *Object-oriented Modeling and Design*, Prentice Hall, 1991.

[6] Gamma E. et al., Design Patterns: Abstraction and Reuse of Object-Oriented Design (ed Oscar M. Nierstrasz) pp 406-431, Proceedings of ECOOP'93, Kaiserslautern, Germany, 1993.