Dealing with Non-toy-sized Projects in an Early Phase of Computer Science Curriculum - Made Possible by Object-Orientation

Klaus Bothe
Department of Computer Science
Humboldt University Berlin
Unter den Linden 6
D-10099 Berlin

Abstract

Software engineering courses and projects generally are offered at a graduate level whereas it seems to be valuable to start with them earlier. We report on a project in an early phase of our computer science curriculum, which is accompanying a compiler course. It is non-toy-sized and based on object-oriented concepts that are suitable to cope with the complexity involved in our project.

1 Introduction: The problem and our approach

Basic principles in software engineering education

There is a consensus of opinions concerning the following basic principles of education in software engineering (SE):

1. SE is one of the important disciplines of computer science.
2. Students should be taught SE concepts from the very beginning of their studies [8].
3. An integration of SE principles into existing courses may be superior to offering separate SE classes [10].
4. Modern concepts of SE, like object-oriented (OO) techniques, should be taught rather than traditional ones [6].

Situation in the first semesters

> "Undergraduates in most institutions are being prepared to think about algorithms and writing small programs, but they do not learn enough ... about the complete software development lifecycle" [10].

To put the above principles into practice there are a lot of difficulties to be overcome showing the relative contradictory nature of the above demands. In particular, we have to analyse the situation normally found during the first semesters to derive the proper methods to be applied:

- Students are still at the beginning of their studies and have only little skills and little practice in dealing with more complex software.
- Consequently, the university staff is confronted with motivation problems in applying SE principles.
- There is often only little time reserved for software projects during the first semesters.

The approach: Deal with complex object-oriented software

> "Students do not have a realistic idea of just what large software systems are ... Students should be made aware early on that 'real' software products can be hundreds of thousands or even millions of lines long" [8].

To overcome motivation problems and to build basic abilities in SE, dealing with more complex software in students' projects could be the right way. The term 'complex' should be attributed to 'comparably complex' with respect to the possibilities in the education process. Thus, hundreds of thousands of lines are not realistic, but some thousands may be a good choice in undergraduate education.

Object-orientation is especially well suited since it is one of the most modern and powerful concepts in the production of large software. The benefits of object-orientation of are useful in practice and will also be found useful for our purposes in undergraduate education: Components of an OO software system can be introduced stepwise; and the activity of the students can be concentrated on components comparably independent of the rest of the system.

In this way, students can be made sensitive to problems connected with complex software. Considering the resulting improvement in motivation, students can be more involved in solutions offered by SE concepts.

SE concepts in a compiler project

> "Compilers provide a natural application of the object modeling technique" [9].

With this concept in mind we decided to integrate SE concepts into an existing compiler project in the third semester. Compiler projects accompanying a lecture in compiler techniques have two main options:

- Either solve a lot of single independent exercises;
- or several exercises are concerned with an integrated compiler project.

We decided for the latter way in the following specialized form: Students were given a complete compiler for a subset of the programming language Pascal covering more than 2000 lines of source code. Its structure was modularized using OO techniques. The students' task was to modify and to extend this compiler by new constructs not implemented yet. The individual exercises were concerned with particular compiler components. Altogether the students' activities typically included maintanance tasks, which are important activities of programmers in practice too.

By using this project the OO compiler structure could be discussed, and benefits of SE design and implementation principles could be emphasized (e. g. reuse of components, modifyability ...).

2 Design of an OO software architecture of our project compiler

Coping with compiler complexity

Compilers are complex software systems and, thus, have to be handled in accordance with SE principles. Compiler complexity can be conquered in two orthogonal ways by decomposition:

- functional decomposition: multi pass compilers (devide the transformation process from source to target into a number of passes with a data interface, i. e. an intermediate language)
- object-oriented decomposition: structure each single pass into a number of objects solving one particular subproblem

Consequently, a one-pass-compiler is especially convenient to demonstrate an OO decomposition since there is no functional decomposition via passes. Consequently, we decided to use a one-pass compiler.

Software architecture of our project compiler

Our one-pass compiler translates a non-trivial subset of Pascal into a virtual code (similar to P-code). It is modularized into a number of interrelating objects which led to a software architecture indicated in figure 1. Here we use the Booch notation of object diagrams [1]. In detail, objects are named and included in a rounded box (ellipses we use for objects reused from the C library). Lines denote relations between objects. A small circle at the end of a line marks the 'uses'-relation: The object connected with a circle at one end of a line uses services (operations) offered by the object at the other end of the line. In other words: It sends a message to the other object.

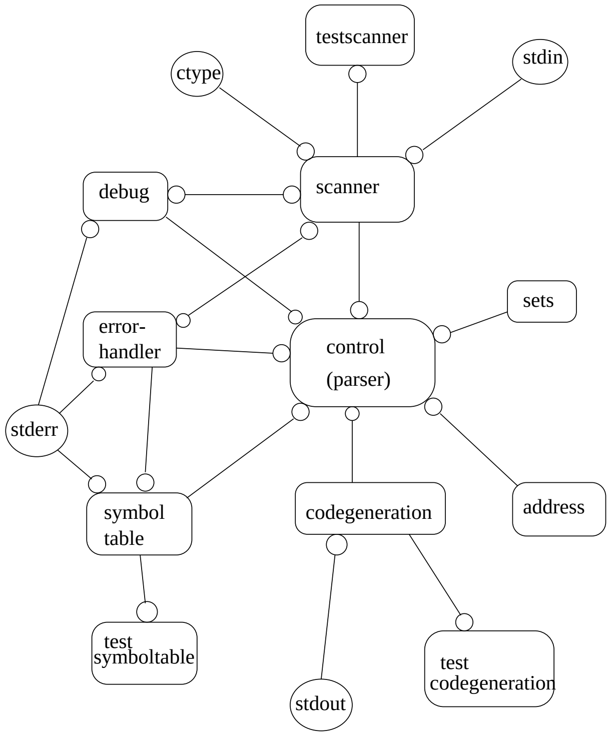

Figure 1: OO system structure of the compiler

This architecture comprizes

- 5 user-implemented 'real' objects: scanner, symboltable, sets, codegeneration, address
- 3 functional objects: control (parser), debug, errorhandler
- 3 dedicated test programs: test scanner, test symbol table, test codegeneration
- 4 objects derived by reuse of the C library: stdin, stdout, stderr, ctype.

We distinguish 'real' objects as a collection of related operations acting on common data from functional objects that consist of functions with similar tasks without common data (e. g. the set of parser functions in a recursive-descent parser). The latter type often is also treated as an ordinary object, it should be distinguished, however, from a methodological point of view.

With this domination of 'real' objects we could speak of an OO architecture.

Our objects are designed to be problem-oriented: Subtasks of the compiler are implemented by objects. For example, the functional object control (parser) implements, by a collection of parser routines, the LL(1) recursive-descent syntactical analysis. The 'real' object symbol table is a collection of symbol table entries and a set of access routines to these entries.

3 Illustrated SE concepts and benefits of an OO architecture

There are a number of important SE concepts involved in our project. We will discuss each of them - in general and, in more detail, how we used the potential of our project to stress these concepts.

Software architecture quality

> "Few interfaces: Every module should communicate with as few others as possible" [7].
> "Small interfaces (weak coupling): If any two modules communicate at all, they should exchange as little information as possible" [7].

Have a look at our system structure visualized in figure 1. With this object model we could illustrate the two principles stated in our box very well. The control module (parser) has used the services of nearly every other module - it is the central module. On the other side, the connections of the other modules are really rather minimal. For example, the codegeneration module only has relations to the control module.

Object-orientation and modifyability

> "Change is usually not a serious issue with small programs ... As programs grow bigger, they become harder to adapt" [7].

The techniques to enhance modifyability may be introduced on small examples. Their relevance only becomes clear, however, for larger projects [7].

In our project, we made changes like the change of symbol table representation (instead of linked lists take hash tables or sorted binary trees). Since the symbol table interface remained unchanged, the rest of program also could remain unchanged. Only the symbol table implementation had to be changed, which was a convincing fact for the students.

Reuse

> "A high degree of reuse means that far less code must be written for each new application" [1].

By this thesis of Booch, every student will be convinced. There are several possibilities to illustrate reuse of compiler components. For instance, the scanner component is reusable in compilers based on different parsers: LL(1)-parsers working according to the recursive descent method as well as table-driven, and LR(1)-parsers.

This also applies to the symbol table which can be reused in one-pass compilers for block structured languages.

OO development strategy and testing

> "The implementor must have a well-planned implementation schedule for the various subsystems. The system must continue to function as the various systems are added." [4].

We introduced the whole compiler system stepwise: Each component of the compiler could be implemented and tested nearly independent of the others (module test), and then it was integrated into and tested in the existing system (integration test). The module test had been performed by an additional test program from where it was invoked.

This procedure may require the simultaneous development of a number of components if there are cyclic dependencies between them.

In our case, the three components scanner, error_handler, and debugger were the first to be programmed and tested, after that the parser (using the scanner) could be implemented. Next, the symbol table was implemented and tested separately, and finally used by the semantical analysis called by the parser control component. The last step was the incorporation of code generation.

During the whole development process, while new components had been added, all but the control component (parser) remained unchanged.

Problem-oriented debugging tools

> "Instrument the program for debugging and performance monitoring" [9]

Besides problem-independent general-pupose debugging tools, dedicated debuggers may be very helpful to find errors in special projects. The programmer has to plan and implement them as part of the program. We did so by giving the students the final solution.

For example, the debugger of the scanner decoded for each symbol read the internal representation (which is a collection of coded numbers) of this symbol: Thus, students could observe whether the scanner really recognized the proper symbol. Later, the interrelation of scanner and parser was made visible. As a next problem, the correct implementation of the symbol table was made obvious by decoding its contents.

4 Basic project activity: maintenance

Software maintenance - the main activity of a practicing software engineer

> "Practicing computer scientists ... deal with ... tasks that involve modifying software systems rather than developing them" [8].

In our compiler project students have to understand parts of the compiler given to the students before they can solve their tasks of modifying and extending the software.

Some students critisized our approach with the following arguments:

- It would be more interesting and more valuable to develop the whole compiler from scratch.
- Understanding the software takes too much time.
- The effort to test the modifications is too expensive.
- The essential task of modification is comparably easy.

These reactions are typical and intersting, and should not be ignored. To cope with these arguments we told the students that the main activity of a software engineer in practice is just what we did: Maintanance, i. e. modifying existing software rather than developing new one.

Within this maintanance process the separate activities are understanding code (50% of the whole maintanance process), plannning the modifications (10%), performing the modifications (15%), and testing them (25%) [3]. These percentage relations just correspond to the observation of the students, and so they could be more convinced that they are in line with practice.

The problematic of maintanance projects

> "Object-oriented analysis and design is certainly a dynamic process. Displaying a 'final' state often gives the learner a false expectation of being right on the first try" [5]

From the educational point of view we are faced with the problem that our project compiler is given to the students in a final state whereas it was developed in a longer design and implementation process. This is just a general problem of students' maintenance projects, and, thus, we have to address this issue.

We did two things to prevent students from misunderstanding the software development process:

- We emphasized that our software has emerged from a longer development process. In particular, it is important to stress that modules (objects) never can be desinged the first time: "We must use them in context first, and then improve them over time as we learn more about the behavior of the system" [1].
- In particular, it was useful to give examples how our object classes evolved, e. g. how their interfaces changed.

  After this, a discussion about alternative object structures can be added which would lead to alternative (or even improved) system architectures.

5 Discussion

Education phases in SE

> "Object-oriented concepts can and should be applied throughout the entire software life cycle" [9].
> "Object-oriented technology is more than just a way of programming. Most importantly, it is a way of thinking abstractly about a problem using real-world concepts, rather than computer concepts ... Coding is the last stage of development" [9].

A single course at the beginning of the study cannot cover all concepts of SE. In our course, we emphasized on OO software architecture covering aspects of modularization, information hiding, minimal interface, software reuse, team work, modification, and the important task of maintenance.

Not included were the early phases of software development: analysis and design of software systems.

However, in our opinion, analysis and design should be tought or trained only after students had studied examples of good software architecture. Thus, SE education should be in two phases the first of which we included in our course project.

1. **phase** in the curriculum: study good software architectures (compiler, 3. Semester)
2. **phase** in the curriculum: students perform the design themselves (later)

Suitability of compiler projects for SE

> "In the past, computer science curricula have dealt with the need for complex system experience by offering project courses in compiler construction or building small operating systems. We believe that many software engineering problems are not touched upon with these courses" [2].

Some authors claim that projects in compiler, operating systems, and other well-established fields of computer science are not suitable for SE education (especially projects) [2]. Due to the well-established theory and practice there is a stringent final structure of the software product, and, thus, the analysis, design, and implementation process is too straightforward, not typical for practice.

However, with our project, we are still in the first phase mentioned above: Study examples of good software projects. In this phase, building on a well-established domain is just an advantage, since good architectures can be suggested by the teachers.

Suitability of compiler projects for the OO method

> "A compiler is almost a pure function ... The functional model is the main model for such programs" [9].

With this thesis, Rumbaugh takes the functional model to be the relation between input values and output values in a computation [9].

If this was the truth, the OO method would not be very conveniently taught by compilers, since the main decomposition of a compiler would be a functional decomposition into passes, unsuitable for object-orientation. However, we showed that a compiler comprizing only very few passes (only one is the best) is an ideal entity to be decomposed by objects.

Summary

"Study good examples of software systems" [10]

In an early phase of a computer science curriculum we organized a project accompanying a compiler course. The software concerned covered more than 2000 lines of source code. Considering the limited time frame, this amount of software would not have been managed by the students and the university staff without a sensible modularization of the whole software.

Thus, we used the potential of this compiler project for the education in software engineering. Normally, the students' motivation often suffers in SE education which could not be observed in our case. On the contrary, in combination with an established field of computer science, the students were open to SE concepts.

References

[1] G. Booch, "Object-Oriented Analysis and Design", The Benjamin/Cummings Publishing Co. 1994

[2] B. Bruegge, "From Toy System to Real System Development: Improvements in Software Engineering Education", SEUH '94

[3] A. Frazer, "Reverse Engineering - hype, hope or here?", in: Software reuse and reverse engineering in Practice, Chapman & Hall 1992

[4] J. Holmes, "Building Your Own Compiler with C++", Prentice-Hall 1995a

[5] J. Holmes, "Object-Oriented Compiler Construction", Prentice-Hall 1995b

[6] M. Manns and G. Puhr, "Object-Oriented Technology Education and Training: Bridging the Gap Between Academia and Industry", in: 8th SEI SCII Conference 1995, LNCS 895

[7] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall 1988

[8] J. Prey, J. Cohoon, G. Fife, "Software Engineering Beginning in the First Computer Science Course", in: 7th SEI SCII Conference 1994, LNCS 750

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall 1991

[10] R. Tewari, F. Friedman, "Integrating Object-Oriented Software Engineering in the Computer Science Curriculum", in: 6th SEI SCII Conference 1992, LNCS 640