# Four Steps and a Rest in Putting an Object-Oriented Programming Environment to Practical Use

GERHARD MUELLER & ANNA-KRISTIN PROEFROCK

Nixdorf Software Engineering GmbH
email: unido!nixpbe!muellerg.bln
email: unido!nixpbe!proefrock.bln

*ABSTRACT*

Designing and developing an object-oriented programming environment is an exciting task. Putting it into practice is a different matter entirely. We did both. This paper reports on the achievements and setbacks encountered in putting A.S.E., an object-oriented programming language and associated environment, to practical use in application software development projects. Although the experience outlined is based on a limited number of projects of various sizes, it might be expected that object-oriented programming is well-suited for modelling a wide class of applications problems. The authors conclude that as increasing use is made of the object-oriented approach, so too will the traditional development cycle see major changes which will not only bring forth better applications, but also lead to problems in moving traditional programmers into the object-oriented world.

## Introduction

Nixdorf founded our group some five years ago with the intention of developing an advanced programming environment for use in general application development. Our work so far has resulted in the design and implementation of an object-oriented programming language and an associated environment called A.S.E. (Advanced Software Environment). The language emphasises data abstraction and supports persistent objects. The environment provides the basic tools, as well as those mechanisms particularly useful and necessary for the development of large-scale industrial software systems.

At the time of writing this paper, the A.S.E. system has been in pilot use for nearly one year and, during this period, has been applied in the development of a number of application software systems. Although our group is a pure development group, we were also involved in these pilot projects. This took the form of providing training for the pilot project team members, as well as giving at least partial support during the design phase.

Nevertheless, our group was required to fix bugs and implement some additional basic features related to pilot requirements.

The paper reports on our experience in putting object-oriented programming to practical use. Of course, this experience is limited to our approach and our fields of experience only. In addition, the presentation reflects more the system developer's point of view than that of the sociologist, a view which would perhaps be more suitable in this particular case.

We will start our report by describing in a little more detail the general ideas and features of A.S.E. We will then focus our attention on the application projects themselves, their contents, size and status. In connection with this basic description, we will outline our experience in detail. On the basis of this experience, we will finally draw our conclusions and present some of our ideas for future work on the language and its environment.

## A.S.E. in Brief

A.S.E. stands for Advanced Software Environment and consists of an object-oriented programming language and an associated programming environment. The language supports single inheritance and emphasises data abstraction, strong typing and persistent objects. Persistent and shared objects are realised by integrating the language into a standard relational database system [DDB/4 '88].

The existence of a database system is largely transparent for the application programmer. In particular, this means that A.S.E. does not provide any special data definition language; all database manipulation is performed by either the translator system or the runtime system. The programmer only defines classes and may declare some of them as being persistent, which implies that for exemplars of this class the instance variables are stored in the database. For most applications, persistent objects must be uniquely identifiable throughout the system. Thus, the language supports nominators which serve as unique handles to exemplars. Nominators are mapped onto primary keys of the underlying relational world. To control the sharing of objects, the programmer may define some class operation as behaving exclusively. This implies that concurrent accesses to exemplars are blocked. Transaction control is solely handled by the runtime system [Mueller, Proefrock and Schiewe '88].

The runtime system is constructed as a virtual machine. In contrast to the Smalltalk approach of a virtual machine [Goldberg and Robson '83] we support a reduced instruction set. The instruction set covers a little more than 50 instructions and is supported by a dedicated stack model for the invocation of operations and the evaluation of expressions separately. We spent a considerable amount of effort in tuning its implementation in order to yield sufficient performance figures.

The programming environment provides the basic tools, such as a compiler, linker and symbolic debugger, but also some 30 predefined classes. The environment additionally supports a specific application framework in order to further streamline the

development of a wide class of information-system-related applications.

The application framework mainly consist of a user interface management system and a filter/browser system. Both components can be integrated into any application without additional programming effort. The user interface management system supports the usual mechanisms, such as overlapping windows, forms, pop-up menus, integrated (context-sensitive) help systems etc. As outlined above, objects are transparently mapped onto a relational database system. Due to the more complex structure of objects, this mapping is usually a one-to-n mapping. Thus, a comfortable query facility capable of tracing the objects of the application down to flat relations is needed on top of the application. The filter/browser system offers these services and acts as a comfortable end-user tool for querying the contents of the database in an object-oriented and interactive style [Weber '88].

A.S.E. is implemented on top of the UNIX operating system and may run both in a client-server and in a more classical time-sharing environment.

## Fields of Experience

Until now, A.S.E. has been applied on ten pilot projets, five more than were planned initially. Only real-world applications were chosen for pilot purposes. We use the term 'real-world application' to characterise software systems which are initiated with the aim of using the resulting system in an operational manner.

The target of all our projects is the development of information systems (IS). The following table provides an overview of the projects, their contents and their status:

| project | status | py | client | contents |
|---|---|---|---|---|
| WiMi | in use | 1 | gment | IS for economic promotion |
| DCV 1 | in use | 1 | gment | IS for emergency and disaster aid |
| 2 | underway | 1 | gment | IS for general administration |
| Gerd | underway | 4 | gment | IS for the management of district council meetings |
| Trips | underway | 5 | indst | IS for travel agencies with integration into booking networks |
| Champs | underway | 30 | indst | IS for production planning and control |
| NME | prototype | - | intern | |
| BVA | in use | 2 | gment | IS for the administration of German schools abroad |
| PMV | prototype | - | gment | IS for the administration of party members |
| BWB | underway | 1 | gment | IS for general administration |

Only one project (NME) is an internal one. Eight projects are devoted to direct client orders, while one project is concerned with the development of a standard application

package for production planning and control. Further projects are expected to begin in the short term. The size of these projects varies from one person-year up to 30 person-years for the production planning and control system. The current status of these projects is that three have been concluded and are now in operational use at client sites, six are in various stages of completion, whilst one project has failed.

In our search for pilot projects we did not set our sights on any specific application domain. However, all these projects are advanced and complex office application systems with some degree of consultation and decision support. None of them constitute a classical data entry system. Users of these systems are usually skilled office clerks with some freedom when it comes to decision-making. We cannot offer any sound reasoning for this application domain. We can only expect that object-orientation is considered as some future technology and thus more likely to be chosen by advanced application systems as an implementation platform.

If we take a closer look at the above table, it appears that governmental organisations mainly constitute the customer base for A.S.E. projects, while industrial enterprises seem to take a rather conservative stance by comparison (this tendency also holds true for the projects we expect to start in the short term). In view of our limited range of pilot projects, this concentration could be a mere coincidence. However, it may be worthwhile offering two possible attempts at an explanation. In Germany at least, the majority of governmental organisations have committed themselves closely to UNIX. This commitment itself implies a break with traditional mainframe-oriented technologies. Use of object-oriented programming is therefore only a small step further. Another explanation could stem from the application domains themselves. For typical governmental applications, advanced user interfaces of the type supported by most object-oriented systems are of more concern than the high transaction rates required in a large number of industrial applications.

Before reporting directly on our experience, we would like to provide some remarks on the general project procedure. Every project team was at liberty to decide whether to adopt A.S.E. or some more traditional approach, such as C or COBOL. This meant that we also had to "sell" our approach to candidate teams. Those teams who decided to go with A.S.E. (usually following prior consultation with the client) received a basic training course and partial support by one member of the A.S.E. development team during the design phase. The A.S.E. team member was not called upon to carry out any design activities as such, but rather to prevent the the project team from making fundamental errors.

## The First Step or Learning Object-Oriented Programming

Participation at a training course is usually the first step into the A.S.E. world. Due to the fact that object-oriented programming is a comparatively new technology at Nixdorf, the A.S.E. team is called upon to hold basic training courses also. So far, we have given a total of eight courses and have trained more than 70 programmers. Most of the

participants of our courses were Nixdorf programmers (more than 70%) with a more or less solid background in C or COBOL. The other participants came from cooperating software houses (some 20%) and also a minority from clients. In particular, some participants sent by clients had only casual experience in programming, while their knowledge of system organisation and the application domain itself was quite extensive.

None of us has any sound experience in teaching. For this reason, we extended our team by recruiting a (non-programming) psychologist with the underlying objective of achieving some paedagogical support. In spite of all this, the first two courses at least met with a disastrous end. When conducting these courses, we fell back on our own experience in learning various procedural programming languages. Thus, we focused on programming in the small, emphasising on language syntax and programming. In the meantime, we agree with Knudsen [Knudsen and Madsen '88] and our psychologist [Wittstock '88] and now place equal emphasis on object-oriented design and theory. The basic training course lasts one week. A day-and-a-half is solely devoted to teaching object-oriented design and the basics of the paradigma. The remaining time is spent on practical exercises.

For exercise purposes we use a small but complete application. A written (verbal) specification is delivered for this application. On the basis of this specification, a design must be developed and agreed upon by the whole course. In a second step, this design must be implemented cooperatively.

We regard cooperation as absolutely vital if object-oriented programming is to succeed. Unfortunately, cooperation is something totally alien to the majority of programmers; they are used to working on 'their' program, which is regarded as a kind of intimate possession. Cooperation is only a matter of some final integration task. As far as our courses are concerned, this means in actual fact that classes which have to be implemented are distributed between the course participants. Thus, the participants have to proceed through implementation, interface conferences and some formalised release procedure. In order to speed up implementation, we also supply a skeleton implementation of the application (it is part of the teacher's instinctive feeling to influence the group's design to a certain extent so that it conforms more or less to the predefined skeleton).

As regards the progress of learning, a distinction could be made between roughly three groups of participants.

People who have learned programming on some theoretical level at university have almost no problems whatsoever when it comes to learning and using the object-oriented approach. If any problems are encountered, then these centre on problems of cooperation.

Casual programmers with a solid application background also seem to be quite successful when it comes to in using this approach as some kind of design vehicle. Of course, this group runs into problems during implementation (scope rules, parameter passing etc.). However, problems of cooperation are non-existent.

We encountered the majority of problems with the bulk of traditional application programmers who learned programming by way of a process of trial and error. The difficulties which faced us elsewhere occurred when designing the application system in

an object-oriented style, when implementing classes through small and clear operations and, last but not least, when relying on the implementations of other participants. On the basis of this experience, we expect severe problems for the future in moving the bulk of traditional programmers into the object-oriented world. In our view, the switch from procedural to object-oriented programming will cause more problems than the swing from assembler to high-level programming some fifteen years ago. Although this did not change the way programmers proceed when it comes to solving problems, the next step will certainly do just this.

## The Next Step or Writing the First Application

After attending a basic course, participants usually get down to working on their application. Design is followed by implementation, where initial progress is swift and enthusiasm is high. After only a few weeks, some teams reported that their application system was nearing completion - despite the fact that the time required for this was a great deal less than the time envisaged. Things usually came to a grinding halt after six to eight weeks. During the course of problem analysis, it transpired that the teams in question had used object-oriented programming as a kind of workhorse for prototyping. The result of their development work was only the skeleton of a huge application system. When it came to implementing the details, this skeleton turned into something of a maze in which they became hopelessly lost.

We discovered two phenomena in conjunction with this collapse. Firstly, programmers encountered severe problems in using predefined classes and, secondly, we came to fully appreciate the value of a monitor facility in object-oriented programming.

As in the case of most object-oriented programming languages, A.S.E. supports only a small set of built-in operators, but contains some 30 predefined classes. From a language designer's point of view an elegant solution, but for the programmer it is quite cumbersome to learn to use these classes in a correct and efficient manner. We have learned from this experience and have recently started offering an advanced course on A.S.E. This course also lasts one week and deals in detail with the use of predefined classes and other implementation hints. Participation in this course is strongly recommended for at least one member of every project team.

We also learned to fully appreciate our debugger [Brodde '87] which allows its user to navigate through the application system in a comfortable and interactive manner, to follow the flow of control at source-code level and on the level of the virtual machine, to inspect the value of local (reference) variables as well as the value of instance variables, to follow inheritance and so on. We found that programmers not only used the debugger to track dow bugs, but mostly to learn about the functionality of used classes or inherited classes. Although our documentation has not yet reached its final state, we believe that even when a final version is available future programmers will continue to place more faith in monitor facilities such as these than in any written description.

### One Step Back or Reimplementing the Application System

Nearly every application system in our sample had to be redesigned at least once. We personally believe that every substantial part of a software system has to be written at least twice to reach the level of maturity necessary for products. However, rewriting is usually the result of having learned how to do it better, while in our projects this became essential since further progress would otherwise have been blocked. The reasons for this necessity are twofold: one is based on overestimating inheritance and the other on insufficient consideration being given to reusability.

Inheritance in object-oriented systems is both a classification scheme as well as a mechanism for code sharing [Madsen '88] [Snyder '86]. In the latter case, inheritance must be used very carefully. Our project teams overestimated the value of inheritance and tried to apply this concept to minimal code sequences also (programmers usually regard themselves as being lazy). After a short time, the application system was not maintainable at all. Surprisingly, redesign did not change the classification scheme but decreased the level of code sharing considerably.

During our training courses we place great emphasis on reusability. Thus, people usually started to implement their classes with reusability in mind. But it is by no means sufficient to have reusability merely 'in mind' because the first attempt to reuse some class usually resulted in a reimplementation of major code parts related to the specific requirements of the client program. Unfortunately, however, the next client program brought with it the same degree of effort. Before iterating in some endless loop, people decided either to block their class from general reuse or to throw it away and to start from scratch. The search for reusability must be part of the general design; later on, it is practically no longer achievable. We believe also that the development of code which is generally reusable is a relatively expensive task and should therefore be planned carefully in advance.

Fortunately, our project teams did not worry too much about the necessities of starting again. They had learned from their experience that this work was worthwhile and also that they did not lose too much time in redesign and partial reimplementation.

### The Final Step or Installation and Operational Use

Three of our pilot projects are now in operational use at client sites. For these projects at least we can estimate some productivity figures.

Object-oriented programming is claimed to shorten the development cycle considerably. We can confirm this assumption. Object-oriented programming offers the possibility of directly modelling the real world entities of the application domain. The translation step usually needed for transforming the requirement specification into a system architecture is minimised and may be neglected. Thus, object-oriented programming is much closer to human thinking and, on a level of consultation at least, is well-suited for involving the application expert in the design process.

Of course, we do not have any exact figures on the increase in productivity. These figures would imply that at least one project had been developed parallel to this using a traditional approach - an unrealistic request for projects of this size. In accordance with feedback given by the project teams, we can, at any rate, estimate an increase in productivity by a factor of four at least for this class of information system. This increase is not, of course, due only to object-oriented programming, but also to the environment which supports transparent handling of a database system and the use of a user interface system. For the future, however, we may even forecast further improvements due to the learning curve of programmers and the existence of reusable classes. An example which clearly illustrates these expectations is our very first application system which was written by one member of the A.S.E. team. The time required for this project was estimated to be one person-year using traditional methods. We were able to shorten this time by some magnitudes. System design was carried out in cooperation with the client and took two weeks. Implementation took a further four weeks. After seven weeks in all, a first version of the system went into operational use at the client's site.

Fortunately, we were unable to confirm another assumption in relation to object-oriented programming. Object-oriented programming is often considered as some kind of vehicle for initial prototyping. This is due to a variety of reasons; some of them are related to the general language design which often focus on the latest progess of object-oriented languages instead of professional use, and others are related to the insufficient implementation of some systems. At the language level we support data abstraction and strong typing. As far as implementation is concerned, we used proven techniques and spend considerable effort on improving its peformance. We could find neither any loss in reliability nor a non-affordable loss in overall efficiency. Those applications in operational use at customer sites run practically without problems.

However, we were able to follow another advantage of object-oriented programming. These applications run without any maintenance effort. This holds true despite the fact that additional applications have been installed at user sites in the meantime. In particular, we feel that this reduction in maintenance effort is worth even more than any increase in development productivity.

## A Rest on Problem Cases

So far, we have not mentioned any problems as regards our environment as such. It goes almost without saying that we came across problems - more problems than we would have liked. Although most of them were 'homemade' and resulted from our approach, we think that a brief description of these shortcomings would also be of interest to a wider community.

For the class of information systems discussed here at least, support of persistent data, information sharing and transaction control play a major role. Indeed, none of the project teams would have chosen A.S.E. without integration into a database system. Programmers are usually enthusiastic about the ease with which persistent objects can be

handled. In particular, this applies for those programmers who have experience with the problems which usually arise in the development of an application system on top of a database system from previous projects. There is no need to worry about schema design, transaction control, deadlocks and the like.

Unfortunately, nearly all our pilot projects needed to access the database directly. This need is usually restricted to a very small part of the overall application system. The reasons for this vary. In certain cases, they are due to incorrect design, e.g. not object-oriented, sometimes to performance considerations. However, A.S.E. does not provide any programmatic interface to some kind of object-oriented database system [Dittrich and Dayal '86] [Bloom and Zdonik '87] [Andrews and Harris '87]. The only solution is to use C on top of SQL. However, it is then cumbersome to interpret the data structures generated by the translator system, to maintain reference counts and so on. In practice, this implies that applications will have to make do with A.S.E. - or quit!

A second problem we encountered with regard to persistent objects is based on our close committment to DDB/4 as the sole database system. Although DDB/4 is, as internal benchmarks show, at least competetive to any relational database system on the market, clients will not accept any programming environment which forces them to use one specific database system. The choice and the use of a database system is a major investment and is considered to be of long-term strategic significance. We think that for the majority of clients this decision is of even greater significance than any committment to a particular operating system. Thus, a programming environment must be open to any database system in order to be widely accepted.

In general, we may state that our approach is not sufficiently open - a statement which holds true for some object-oriented environments, such as Smalltalk, for example. Due to our implementation of the runtime system as a virtual machine it becomes quite complicated to integrate some foreign application system into an A.S.E. application system. The virtual machine runs as a UNIX process. The only way to integrate a foreign system at present is to encapsulate it by an A.S.E. class interface and then implement this interface using UNIX mechanisms, such as pipes or shared memory - an approach which presupposes that the foreign application system also constitutes a process in itself. It is practically impossible to extend the system using a simple C function, for example, without detailed knowledge of internals. In general, we can assume that our choice of a virtual machine was not one of our best decisions. The virtual machine of course offers a high level of flexibility, thus providing the basis for the quality of a number of tools, such as the debugger, for example. However, it limits our performance, restricts our openness and requires a great deal of building and maintenance effort.

We stated above that we have so far not encountered any real performance bottlenecks with our pilot projects. Partially at least, this stems from the low transactions ratio we fortunately found in these systems. But nearly every project in our sample had some limited but essential need for system programming, such as integration into public networks, for instance. Usually, we ended up by having to provide at least the basic functionality as an extension to some predefined class. This process is hopefully finite and we may expect that we are now more or less able to fulfil the usual requirements for

system programming of the majority of applications.

However, it nevertheless seems to be worth emphasising a high level of performance, for example by native code generation. The programming language C did not succeed as as a result of its conceptual elegance, but rather due to its ability to support programmers in writing programms, thus allowing them to be executed almost as fast as hand-coded assembler programs. Similar performance would at least offer the argumentative base to win over C programmers. But it would also offer a means of bootstrapping the system, an approach which would be equally elegant and concise, and which would solve most problems with regard to openness.

Another problem we encountered is due to our disregard for standards. As with most projects which were initiated some years ago, we did not give sufficient consideration to raising of at least de facto standard software components. Thus, we implemented the whole system from scratch. This is not only an obvious waste of money, time and energy, but also considerably limits the spread of any system. A user interface management system without X/window as a base, for example, will only find limited acceptance.

Another thing we learned from supporting these projects was that object-oriented programming calls for concurrency. In all projects it took two weeks at most for the first programmer to arrive with the natural requirement to execute some object in parallel. This is even more surprising since this requirement did not stem from any performance bottleneck.

A final problem we encountered is the incompatibilty of requirement specification tools and object-oriented programming. Some of our pilot projects had previously used some standard tool for requirement analysis. These tools emphasise separation of data and code, a feature which is contrary to the object-oriented approach. Thus, none of these projects was able to directly transfer the results of the requirement analysis into design. We feel that the lack of methodologies and tools for the early phases of object-oriented design is one of the most exciting development tasks for the future.

Last but not least one project failed. The reasons for this are quite straightforward. Object-oriented programming requires a much more intensive project structure. There is no notion of programs to which programmers may be assigned and on which they can work in isolation. Everything is a class and may be reused and extended. The tools for ensuring consistency are far from perfect. In spite of this, however, the success of object-oriented programming relies heavily on the personal capabilities of the people actually doing the work.

## Conclusion and Future Work

We can conclude by stating that the success of our pilot projects was quite considerable - otherwise we would not have reported on it. A further sign of this success is that A.S.E. is due to be released for general use on Nixdorf UNIX machines in April '89.

We have learned some vital lessons which we will put to the best possible use in our future work. The majority of these lessons were outlined in the short rest we took to discuss the problems we encountered. The platform for this work will be the ESPRIT Ithaca

project (2121), a joint effort of major European software companies (including Nixdorf, Bull, Gap Sogeti, Datamont, Delphi and Toa), as well as a number of academic institutions (including the University of Geneva, Trinity College Dublin, the University and Polytechnic of Milan, the University of Athens, Forth, Inria and FZI).

The Ithaca project aims to develop an integrated application support system based on object-oriented techniques. This work will focus on the development of arbitrary methodologies and tools for object-oriented design and system construction. In addition to these tools, the system will incorporate a multi-media environment based on ODA/ODIF standards, an object-oriented database interface system based on standard relational database technology, and a programming language which supports active and distributed objects and which is smoothly integrated into the database system interface. The system will be presented using an appropriate user interface system based widely on standard components. In order to validate our approach, the development of application systems will start as early as possible.

The activities Ithaca entails centre on integrating and enhancing existing components and using standards rather than redeveloping everything again from scratch. In this way, we will achieve the ultimate objective of attaining an open and integrated environment. Last but by no means least, we are concentrating on use of the object-oriented style in order to reuse and specialise components, the overall aim being to obtain products of a higher standard.

## Acknowledgements

Design and development of A.S.E. is a team effort. As members of this team, the authors regard themselves merely as its mouthpiece. We extend our thanks to the partners involved at the Ithaca proposal phase who provided us with useful and interesting suggestions, and to Stephen McMahon who helped edit and revise our jumble.



## References

[1] T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 430-440, Orlando, FL, October 4-8, 1987.

[2] T. Bloom and S.B. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 441-451, Orlando, FL, October 4-8, 1987.

[3] E. Brodde, "Konzeption eines High-Level-Debuggers unter besonderer Berücksichtigung des object-orientierten Ansatzes A.S.E.-OBJECT", Diploma

thesis, Technische Universität Berlin, 1987.

[4] "DDB/4 Reference Manual - Concepts and Facilities", FZI-No. 10.237.00.7.93, Berlin, 1988.

[5] K.R. Dittrich and U. Dayal, *Proceedings of the International Workshop on Object-Oriented Database Systems*, IEEE Computer Society Press, 1986.

[6] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[7] J.L. Knudsen and O.L. Madsen, "Teaching Object-Oriented Programming Is More Than Teaching Object-Oriented Languages", in *Proceedings ECOOP '88*, ed. K. Nygaard, Springer Verlag, 1988.

[8] O.L. Madsen and B. Moller-Pedersen, "What Object-Oriented Programming May Be - and What It Does Not Have to Be", in *Proceedings ECOOP '88*, ed. K. Nygaard, Springer Verlag, 1988.

[9] G. Muller, A.-K. Profrock and J. Schiewe, "A.S.E.-An Object-Oriented Programming Environment for Industrial Use", in *Conference Proceedings of the German Unix User Group*, Network GmbH, Hannover, 1988, German Version.

[10] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", ACM SIGPLAN Notices Proceedings OOPSLA '86, vol. 21, no. 11, Nov 1986.

[11] M. Weber, "Entwurf eines Filter/Browser-Systems", Thesis, Technische Universität Berlin, 1988.

[12] M. Wittstock, "Teaching object oriented programming to application programmers and endusers", Proceedings of the second MACINTER Conference, Elsevier Science Publishers B.V. (North Holland), 1988.