Session 16.1
OBJECT ORIENTED DESIGN PRINCIPLES
O.M. Nierstrasz (CUI)

Introducing Dr. O.M. Nierstrasz

Unlike the previous two speakers, Dr. Oscar Nierstrasz is a pure computer scientist who earned a PhD in Computer Science at the University of Toronto in 1984. His thesis dealt with the analysis of the behaviour of systems with a large number of correlations between procedures and processes. He then continued with post doctorate work on the design of a general purpose object oriented programming language that integrates object oriented features with concurrency, persistence and strong typing. His interests remain in the object oriented field, especially concerning the semantics of concurrency in object oriented languages, and advanced tools for interactive construction of OO applications.

Comment at editing: Dr. Nierstrasz's lecture did actually deal with object oriented design principles. But since the material presented is already covered in the article he published in the proceedings of the 1986 CERN School of Computing (CERN 87-04), he preferred to publish here a more or less linearly readable Glossary of Object Oriented Concepts, which makes good complementary reading.

A Glossary of Object-Oriented Concepts

O.M. Nierstrasz

University of Geneva
Centre Universitaire d'Informatique
12 Rue du Lac, CH-1207 Geneva, Switzerland
oscar@cui.unige.ch, oscar@egeugeq.bitnet
mcvax!cernvax!cui!oscar

1 Introduction

Object-oriented programming is a paradigm for organizing software into reusable components that combine data structures and the procedures that can be used to manipulate them. Objects go beyond traditional structured programming by providing more powerful mechanisms (mainly class inheritance) for reusing existing software components to generate new ones.

Historically, objects and object classes were introduced in Simula, an extension to Algol to support the programming of computer simulations [Birtwistle, et al. 1973]. During the 1970s, these ideas were further exploited in Smalltalk, a fully object-oriented language developed at Xerox PARC [Goldberg and Robson 1983]. The Smalltalk experiment demonstrated that reusable software objects could be applied not only to the development of simulations, but to application development in general.

During the 1980s, there has been an explosion of interest in object-oriented languages, and in object-oriented extensions to existing languages (Lisp, C, Prolog, etc.) in an effort to increase the power of these languages, and especially to reduce the cost of developing applications by making software more reusable, and hence more reliable.

This paper presents the most important ideas contributed by this field in the form of a glossary, starting with the fundamental concepts, and proceeding to some more advanced topics. The glossary is intended as an introduction and a guide rather than as a complete "encyclopedia" of object-oriented concepts. Since the area of object-oriented programming is rapidly evolving, this presentation should be viewed as no more than a "snapshot" of current developments. The bibliography contains a representative sample of the literature in this field.

2 Glossary

The entries in this glossary are presented in a logical progression so that they may be read linearly (rather than in alphabetical order). Cross-references are indicated by terms in italics. Related entries are generally grouped closely together in the text.
Object: An object is an encapsulation of data and behaviour. Objects typically belong to an object class, a set of instances with the same internal structure and the same external interface. The persistent data of an object are usually represented as a set of instance variables. The interface is usually a set of functions and procedures that can be invoked by sending a message to the object. Upon receipt of a message, the object then selects the method (i.e., procedure) for handling the message. An example would be a circle object with instance variables x.position, y.position and radius, and with operations move and display.

Instance variable: A complex object contains persistent data named by a set of instance variables, each of which identifies another object. A primitive object (like an integer) has no instance variables. Object-oriented extensions to existing languages, such as Lisp with flavors [Moon 1986] and C++ [Stroustrup 1986], often make a distinction between the "objects" of the extension and ordinary data values in the base language. In this case, instance variables may identify (unencapsulated) data rather than objects.

Message: Message-passing is the paradigm for communication in object-oriented languages. This is meant to emphasize the fact that objects properly encapsulate data and operations: in traditional procedural languages the application selects and applies procedures to manipulate passive data; in object-oriented languages, we send a message to an object, which itself selects the appropriate method to execute. It is not possible to force an object to do something that it is not equipped to do. An object is therefore a data abstraction. More often than not, "message-passing" is actually implemented by procedure calls, although a table-lookup may be necessary to locate the code to execute (see class inheritance).

Method: A "method" is a procedure selected by an object in response to a message. Since the responsibility for selecting the method lies with the object rather than the application, it is possible for two objects to respond with different methods for the same message. See class inheritance and polymorphism.

Data abstraction: A data abstraction packages a data structure and the operations that may be used to access and manipulate the data. Languages that support data abstractions include C [Liskov and Guttag 1986], Modula-2 [Wirth 1983] and Ada [ANSI 1983]. These languages are sometimes classed as being "object-based", since they support objects as a language feature [Wegner 1987]. They are not "object-oriented", however, in that they do not support class inheritance.

Object class: An object class defines a template for creating object instances. For example, a circle object would simply be an instance of the Circle object class. Typically all instances will have private copies of the instance variables defined for the class, and will share methods. In some object-oriented languages (such as Smalltalk), object classes are themselves first-class values – that is, they may be defined at run-time and passed as values.

Prototypical object: An alternative model to object classes for creating instances is to use "prototypes" of objects [Lieberman 1986]. Sharing is accomplished by (feature) "delegation". An object can either delegate messages to its prototype, or provide its own method for handling certain messages.

Class inheritance: Object-oriented languages differ from object-based languages in that they provide inheritance mechanisms for defining new classes from old ones. A subclass inherits all the features (instance variables and methods) of a parent class, and may modify the inherited behaviour by adding new instance variables and methods, or by redefining existing ones. The Circle class might inherit the x.position and y.position instance variables and the display method from the parent class Graphic, and add the radius variable. The Circle class might need to redefine the display method. There are many different approaches to inheritance [Snyder 1986]. Some of the variants include multiple inheritance, partial inheritance, and dynamic inheritance.

Subclass: A subclass inherits features not only from its parent class, but also from its ancestors further up the class hierarchy. Depending on the inheritance scheme adopted by the programming language, the subclass may or may not have direct access to inherited instance variables in its own methods. If it does, the subclass may "violate encapsulation" of its parent. For example, if the Circle object directly accesses its inherited x.position and y.position variables, then we lose the freedom to re-define the class Graphic using polar coordinates for its position. One solution is for Graphic to export an abstract position variable as part of its interface, and hide its implementation in the object class Position.

Multiple inheritance: If a subclass may inherit from only a single parent class, this is termed "single inheritance". With multiple inheritance a subclass may combine features of several parents. For example, the class ColouredCircle may inherit features from both Circle and ColouredObject.

Partial inheritance: In some cases a subclass may choose to restrict its inherited interface. This is possible in both C++ and Eiffel [Meyer 1986], which would permit the class Stack to inherit from LinkedList, yet export only stack operations. This use of class inheritance for implementation sharing is to be distinguished from subtyping. When inheritance is used in this way, it is normal to permit the subclass full access to all inherited features, whether they are exported or not.

Dynamic inheritance: If an object instance can dynamically change its behaviour, we term this dynamic inheritance. Objects of the same class start out with identical behaviour, and eventually diverge as they acquire and discard methods and instance variables [Tsichritzis, et al. 1985; Casais 1988]. Dynamic inheritance of properties may also be a function of the current scope (or environment) in which an object finds itself. For example, a paragraph object will have different display characteristics if it is moved from the body of a document to a footnote.

Object type: A type identifies a distinguished set of values of a programming language, either by enumeration, or by constraint. An object, however, is not a value, but a
thing that has a value (which may change). An object type, then, is a constraint on the set of values an object may have during its lifetime. This constraint may identify precisely one object class, such as Circle, or, if the language supports dynamic binding, and polymorphism it may identify a set of related classes, such as GraphicObject, or any of its subclasses. In this case, the type constraint refers exclusively to an object's interface, rather than its representation.

Subtype: A subtype is a refinement of a type. Any object that satisfies the subtype constraint must automatically satisfy the parent type constraint. For example, if any instance of Circle can be substituted for an instance of GraphicObject (because the interface of Circle includes everything inherited from GraphicObject), then Circle acts as a subtype. On the other hand, when partial inheritance is supported, subclasses may not serve as subtypes: we may not manipulate a Stack using the operations of LinkedList, since this would violate the integrity of the Stack object class.

Polymorphism: A polymorphic operation is one that may be applied to values of different types. For example, a length() function might apply to strings as well as arrays and lists. Since different object classes can support operations with identical names either by inheritance or overloading, object operations are effectively polymorphic. The different forms of polymorphism are well-explained in [Cardelli and Wegner 1985].

Overloading: Object-oriented languages typically provide for overloaded operators, that is, independent object classes may have operations with the same names, though the semantics of these operations may be unrelated. For example, the + operator defined for integer objects may be overloaded to provide a concatenation operator for strings. Overloading provides for a certain notational convenience, and it is essential for implementing subclasses that override inherited methods, but it can lead to unreadable code if used in an undisciplined fashion.

Dynamic binding: A language with dynamic binding allows a variable to be bound to objects of different classes. In a strongly typed language, the variable is constrained to be bound to objects that conform to its type declaration. For example, a variable of type GraphicObject might be bound to an instance of Circle or Spline. Operations invoked on such variables always entail some form of method lookup, since the method to execute will depend on the class of the bound object. Languages like Simula and C++ provide a choice between static and dynamic binding through the mechanism of virtual operations.

Virtual operations: A virtual operation is one for which an inheriting subclass may supply its own method. In C++, the display operation of the GraphicObject class would be virtual, thus allowing subclasses to define their own implementation. There is a method lookup overhead only for virtuals. In Smalltalk, all operations are virtual.

Generic class: A generic class is parameterized by a set of object class names. The array class, for example, is generic, taking as parameters the class of its elements and the class of the indices. Generic classes are similar to generic packages in Ada.

Homogeneity: The principle of homogeneity in object-oriented languages states that “everything” is an object. This principle is adhered to in varying degrees in object-oriented languages. In Smalltalk, for example, object classes are themselves objects, being instances of “meta-classes”. In C++, on the other hand, not only are classes not objects, there are certain primitive data types, like integers and characters, that are not objects (one cannot define a subclass of int or char).

Concurrency: There are basically two standard approaches to providing concurrency in object-oriented languages. Either objects are viewed as passive entities for which accesses must be synchronized via locks or transactions, as in Smalltalk, Trellis-Owl [Moss and Kohler 1987] or Avance/OPAL [Ahlsen et al. 1985], or objects are viewed as active entities passing messages between them, as in POOL-T [America 1987], Hybrid [Nierrastrasza 1987; Konstantas et al. 1988], Act-1 [Lieberman 1987], Concurrent Smalltalk [Yokote and Tokoro 1986] and ABCL/1 [Yonezawa et al. 1986].

Object-oriented database: Most of the work on object-oriented databases (OODBs) has focused on adding the concepts of object classes and class inheritance to databases, rather than on adding database concepts to object-oriented languages. Since structured objects are a departure from flat relational databases, there is to date no standard data model for OODBs. Querying is performed on “visible attributes”, which is somewhat at odds with the principle of encapsulation. Nevertheless, several OODBs have been implemented, including GemStone from Servio Logic [Maier et al. 1986; Purdy et al. 1987], Orion from MCC [Banerjee et al. 1987], and Iris from Hewlett-Packard Labs [Derrett et al. 1985; Fishman et al. 1987].

References


DISCUSSION

after the tutorial of Dr. Oscar Nierstrasz

Stroustrup, RAFAEL: We are using object oriented design and it seems that it all works very well. But there is some type of difficulty that we have encountered. It is like everything is hidden and there is no way in between. When you deal with real time systems and the interrupt routine down at the last level, you have to know exactly how long it can take the processor and in which priority it comes in relation to other interrupt routines that are well defined in the system. You are not so well when you use inheritance, use it for the wrong reasons and get a mess. You want to be able to structure properly, so that when you use inheritance you want to inherit things which are static and you want to inherit generic properties. But people are using inheritance to do very weird things, that it is just inappropriate for their parts, like a car inheriting from its wheels. That is why you need some guidelines for OOP.

Nierstrasz: It is like having a bunch of toys: you can have a lot of fun mixing and combining them, but obviously you are going to get a mess. And I think some of the OOP techniques now, even though they emphasise structure, are being used wrongly. At the OOP conference in Paris last spring, a lot of people were pointing out that many people were not so happy when they use inheritance, use it for the wrong reasons and get a mess. You want to be able to structure properly, so that when you use inheritance you want to inherit things which are static and you want to inherit generic properties. But people are using inheritance to do very weird things, that it is just inappropriate for their parts, like a car inheriting from its wheels. That is why you need some guidelines for OOP.

Wishnura, L.B.: I think the concept of OOP is very new to most of us but I think the idea itself is very old and most of us already did it without noticing. In the world of hardware the same thing has been done for more than ten years already. I think, because you now already have some standard functionalities integrated in LSIs and local intelligence. I think the concept of OOP in software is quite similar to such a hardware trend ten or twenty years ago.

Nierstrasz: Sure, in fact if you read Brad Cox’s book on OOP, he talks about software ICS. You want to package a software object just like you package a piece of hardware and then, when you build a new application, what you do is configuration by taking these parts and putting them together. The problem is that we do not know yet how to design software ICS very well so that they really are reusable. When we want to solve a problem we define certain objects and we implement them. Then somebody else comes along who would like to reuse these objects. But he finds that they do not fit his needs, so he redesigns the objects.

Wishnura, L.B.: In case of the LSI design we can make so many tools, even an expert system. But no such things are available yet for software.

Nierstrasz: Not yet, but in the past one or two years the whole area of OOP has exploded and people are starting to explore these areas. But as you say: the basic idea is not new: people are seeing all those concepts as well. But nobody did all along can be applied in this paradigm while some other things which they were doing were OOP but they were never called that before.

Krawczuk, IMAG: One remark and one question. The remark is as the previous one OOP programming is not quite new. The first OOP language was Simula 67, that makes it twenty years ago. It introduced the basic notions of inheritance, class instances etc., in a primitive form but still the ideas were there. The question is: what is the issue of efficiency versus uniformity? More precisely, it is very nice to treat everything as an object, from the integer to the document of one hundred pages and many chapters, but the problems are quite different: do I want to make my integers persistent? Do I want to garbace collect integers? On the other hand: I would like to do very fast operations on integers but maybe on documents I do not care about operations being fast but I do care about not losing my document.
**Crowley-Milling:** I think also in the accelerator field we have been using objects without knowing it, without calling them objects. If I understand rightly, what we have been calling data modules and equipment modules are in fact objects or very close to objects in your sense.

**Nierstrasz:** Yes a module that has some hidden variables and a state is an object, except that you often cannot do things, instantiation and inheritance, but that does not mean that it is not OO, it just means that it is not as OO as some other things.

**Di Maio, CERN:** Can you make a short resume of the state of the art? Among all the nice features you mention, such as dynamic binding and object persistence in the runtime system ......., what is available right now?

**Nierstrasz:** Do you mean what is available in existing languages?

**Di Maio, CERN:** Yes, in available systems. Which system for instance gives you standard libraries of object persistence, garbage collection and so on? And which programming language gives you the inheritance and multiple inheritance and so?

**Nierstrasz:** There are two basic approaches to OO languages. There are the compiled languages and the interpreted languages. The interpreted ones tend to be a lot more flexible and have dynamic binding and at runtime they may do the multiple inheritance and so on. These are languages like Smalltalk, where everything is an object. You have instantiation, structuring, you have simple class inheritance, you do not have object typing, you do not have concurrent objects, because that was not part of the initial design. However, people have been working on a compiling Smalltalk, two, having type systems for ST and three, having active objects for ST. These are things people obviously want to add. Also making them bigger, because ST was originally a single-user system and one would like to have it distributed. On the compiled end, you have languages like C++ and Objective C. You have languages like Clu and ADA which give you instantiation, structuring, parametrisation, in ADA in particular, object typing, they do not give you inheritance. It depends what you mean by tailoring, whether they give you that or not. ADA gives you concurrency. Clu does not. Concurrency in ADA is not the way that I have described it, it is this rendezvous mechanism and god knows what else. Other languages: recently there is a language called Trellis with Owl, from DEC. Trellis is an environment, Owl is the language and it is Clu-like so it is a data abstraction language. It is compiled, it has multiple inheritance but it does not have active objects and people have been trying to add that. The language we are working on at the University of Geneva has instantiation, structuring, tailoring, parametrisation, multiple inheritance, objects are typed and we want to have active concurrent objects and persistence. But it is a prototype, so we are translating into C which is then compiled. It is not a production language like these other ones I have mentioned. That is more or less the gamut and other languages will probably fall into one slot or the other.

**Cuperus, CERN:** We are experimenting with a sort of hybrid system which off-line provides a sort of object generator by which you can specify your object. In that environment you have instantiation, tailoring, inheritance. Once you generate the object, it is then just a subroutine and you can use it as an object, but as an object it is frozen.

**Nierstrasz:** It seems to be a prototypical object, in a sense, rather than an object class: you create a generic object and then you can have instances of that object.
GENERAL DISCUSSION
on software engineering

Perriglist: The presentations of the speakers raise a lot of questions. I will ask a few myself, just to start the general discussion. They may be of interest to several of us. I ask them as a potential customer but also as someone who is interested in providing real controls for an accelerator complex. My first question is a technical one about the environment. Are real-time control problems such special problems that commonly used methodologies are not applicable? Correlated to this question there is another one, more oriented to the people themselves: [2] Is it appropriate to apply the industrial methods to informatics problems in the experimental physics laboratory working environment? Another question: [3] Is it reasonable to think of increasing productivity by a factor of two in the next ten years? The question disturbed me quite a bit, personally. [4] What can we do with programmers older than say 35? [laughter] Yet another question: [5] Will technology and tools be effectively available to produce information for control without any professional programmers? And the last question, to trigger the discussion: [6] What is the importance and in the medium term must we do in order to reach this factor of two in productivity? Who wants to start, perhaps somebody of the panel?

Harvey BAL: Let me take your first question. I think three years ago, the answer to the question might have been yes. When we started to use the particular methodology that was adopted, we found that we were using techniques, particularly for real-time applications. So much so that we were delving into the literature just to see if we could supplement the pure methodology at the time, which just consisted of data flow diagrams and structure charts, with additional techniques that were around, and try to overcome some of those problems. In fact it turned out to be the same thing in part, learning a lot of new techniques. The state transition diagrams and the entity relationship diagrams would be absorbed into the methodology and I think it is fair to say that most of the original problems have been overcome to a satisfactory degree.

Casaglino CERN: I would just like to answer two questions. What can we do with programmers older than 35? Well, there is a nice book from Toffler that is called The Third Wave that depicts the society after the industrial one and that society there is not anymore a period to study and a period to work but there is study and work, one after the other in an infinite cycle. The answer could then be just that. It is not so much the age of people but rather their will to study something new that is important. As to the question whether it is reasonable to think of increasing productivity by a factor of two, I think it depends first of all on the degree to which we have acquired the tools that are already available. In our case, at CERN, we have not even a faint idea of what are the tools readily existing. So my answer is: we must become aware as fast as possible of these tools available on the market and then get them as fast as possible.

Krasoriat IMA: I would also like to comment on this last question. We should be very careful in such predictions because the experience has shown that there is a cycle of about ten to fifteen years, more likely fifteen, between the time when an idea has been generated and the time it has been turned into a usable product. I have a chart here that I took from a study that has been done on this technology transfer. It shows you different stages. First there is the concept formation of the idea, then you develop it experimentally and you use it internally, then you explore it outside the team that developed it and realize you have real users. Take compiler construction technology; now most of the stages of production of a compiler are automated, but the first ideas were generated in the 1960s when people were talking about grammars and formal descriptions and abstract types and so on. It was like pure mathematics to most of people then. UNIX started in 1971 and think about 15 years to be in real use. Object Oriented programming is not even on the chart. So we should be very careful about this type of predictions.

Sperling CERN: I would like to link the two questions. Luigi Casaglino tried to answer. If we were really successful in introducing new and inevitably complicated techniques which would increase the productivity of our programmers by a factor of two, would we then, by doing so, reduce the age of uselessness of programmers from 35 to 17.5? In that case: have we gained world wide and in particular at CERN and other accelerator communities?

Di Maio CERN: For the two first points, I think that the more important issue in our system is to be able to follow-up the hardware evolutions. What prevents us from moving to new hardware are the layers of software developed on old hardware. Even if the methodologies are not very suited for real time systems, like restrictions we faced in using Petri nets without time extension for describing drivers, they can help to port to new platforms. In our control system we have a lot of low level software without any formal description with which we would have to maintain and when we should not be able to port to another platform. Therefore, using methodologies, even if they do not cover all aspects of a real time control system, are probably a good investment.

MierstraszCU: I want to address the issue: what short and medium term objectives must we define. We do not really have any experience to what degree the object oriented techniques are really applicable, for example, to real time programming. If you remember, a long long time ago people used to say that high level languages would never work because compilers could never generate code fast enough. I think this thing was about concurrency. That has proved completely untrue. Optimising compilers are extremely clever and there is no way that you could hope to get close to what such compilers can do. The nice thing is that when you have a compiler, you do not ever see the code that it generates. In contrary, if you are doing for example you would like your assembler to be relatively readable and that may come from constraint on its efficiency. When it is done by a machine, that is no longer necessary, so they can use any tricks that are available. The same question would arise here. Right now language designers are concerned with things like type checking and with so efficient compiling that they are not even thinking about concurrency. They have not been looking at real time issues, so they have no clue as to what is needed. So in terms of those objectives, I would say we need communication between the language and the Apollo designers and the people who are building real applications so to identify what information needs to be managed for them and what needs to be managed in the end. As to what you throw into a language, it is very nice to have things like tailoring and dynamic binding in a language but if that is going to impose overhead and the applications programmer does not need dynamic binding, you should not enforce mechanisms with overhead. You provide a set of mechanisms in an operating system as tools for programming but you do not impose a policy. The policy is to be determined at the other end where these things are used. So I think the same thing would have to be true in this case. What are the mechanisms that you need? And then you, the user, enforce the policies.
Perriglat: Yes, but programming in the large needs at least some policy. Also, one must do some experiments and a certain prototyping before defining the policy for a large group of users, providing a common end product. That is a difficult point: how to aim correctly a prototype, what are the points to be proved by the prototype or by this approach and what are the elements of the matrix to be used to judge this approach?

Miersztra, CUI: Sure, you want to build software engineering tools, I want to design languages for compilers, so you need to tell me a little bit more about your concern with the policies. I provide the mechanisms. I cannot just sit down and say gee it will be nice if you had this kind of control structure and I give it to you. If those are not the mechanisms you need in order to do programming in the large, well then that is not a very good language concept. So there are two questions. One is where I can provide you, and there is always a choice of possible mechanisms. The second is what it is that you need in order to structure your programs as conveniently as possible. So where I was drawing the line was not the entire software engineering environment but just at the level of language and language mechanisms. Certainly, above that you do want to enforce policy, especially if you want to build large pieces of software using the languages.

Sromewer, Rafał: First about productivity: it can be made better if one uses better tools. In my experience, for example, strictly type checking languages allow the compiler to check more things and so you get the program operational faster. That is one thing. And that can be a great factor in productivity. Second is ADA. I think that academic model doesn't do like it very much, but we have a real positive experience with ADA. It solves an anomaly of problems, it tries to be real time and it turns into GU, except for the class inheritance. We will be able to see, in about one year or even less, pretty good compilers and a pretty good real time response for ADA on the 68K processors. Our experience is very good with it, not on 68K but on VAX. I don't know if it improves the productivity of the software, because the software gets more and more complicated, but it will make available a system which was not available before. My last comment is about what we can do with programmers of more than 35 years. Well, either they get retired or they do not write software or they learn. And this is not only true for software. It is true for Physics and for other fast evolving fields. You ask here the question for software because most of you I think are Physicists and learning all the time new things in physics seems natural to you. But software is a type of burden for at least part of you. But any one who wants to produce the complex software you are producing should not see it as a natural thing. So you must keep learning, there is no other way.

Perriglat: It is the first time that I have heard about some real implementation in ADA. Could you say in a few words what is your problem that you have solved with the ADA approach.

Sromewer, Rafał: Our experience with ADA is divided into several implementations, areas that are not even correlated. The largest experience we have is a project of about eleven man-years and about 30,000 lines of code. From the definition until the end of shipping the product to the user, it took about two years. It was done by a small team and it was implemented on a VAX machine using the VAX/VMS ADA compiler. It was not a real time project and it did not have any real time constraints, but it was a very good experience with ADA in an object oriented design. We are now testing ADA compilers for the 68K family, three of them actually, and they are not too bad. We are testing them for real time embedded computer systems, like your ECA or so, real time and access to I/O and so on. And you need multi-tasking perhaps, etc. It is all there in ADA.

Perriglat: Well for the people older than 35, I agree with Luigi, it is a matter of education. Can I ask professor Krakowiak what are the methods, capabilities, what have we to do for continuous education after university. There are presently numerous seminars available, a lot of companies provide highly priced education, but I am not so sure that it is really efficient.

Krakowiak, IMAG: The first part of the question is: what are the ways of retraining programmers. It is a difficult problem, many people are not quite willing to be retrained. Some people derive a certain sense of power from being the only ones to know the inside of some piece of software. If these people are to use new methods, where information will be more explicitly public, they may lose that power. There is a fascinating book, written in 1971 by Gerald Weinberg, titled “Psychology of Computer Programming”, in which he explains many factors of software productivity and non-productivity. Another blocking factor is that many programmers now older than 35 had no formal training in programming. They acquired their knowledge the hard way and it is difficult to tell them that their knowledge is somehow obsolete. They have to change their thinking habits and their methods of working. It is very difficult to accept that. Unfortunately I do not have a general answer to the question.

Parrhau, CERN: About this question of people older than 35: earlier this week we had already a lecture about management and the speaker insisted on people being trained, though he did not mention how much of the resources of the project should be devoted to the training. I think it would be the duty of the management to make sure that the people are trained continuously. Often training is limited by the degree of personal willingness, but in some organisations training is limited since it is considered as favour. I think training is a must.

Perriglat, CERN: You ask whether it is possible to increase the productivity by a factor of two? We always must look into the future with some experience from the past. Therefore my counter-question is: do we know whether we have gained a factor of two in the last ten years? Ten years ago all people spoke about the magic number of ten lines per day for a good programmer. Do we now produce twenty lines per day? Can anybody answer?

Perriglat: A partial response: yes, I think this increase in productivity has taken place since ten years ago there was a lot of coding done in assembler. Today, we hope, there is not too much of that, certainly not in our group.

Krakowiak, IMAG: I think the question is not the number of lines of code but their expressivity. Today two lines of code may be more meaningful than twenty lines ten years ago and that is what for instance ADA and OOP languages achieve. In two lines of code they express much more than twenty lines ten years ago. A second factor of productivity is the degree in which your software is understandable, maintainable, explicitly documented and its code readable. In these two sense the improvements is far more than a factor of two in the last ten years.

Perriglat: I am afraid we must stop now. We have possibly not received all the answers to our many questions, that would be impossible, but our workshop has given a number of hints and it has certainly given us a lot of new material to reflect upon. Thanks to all of you!