Strategic Research Directions in Object Oriented Programming

Rachid Guerraoui (editor)[1]
Ecole Polytechnique Fédérale de Lausanne, Switzerland

This report is an overview of the field of object oriented programming. It presents a brief historical perspective of the field, then summarizes some major achievements. Four strategic research directions are then presented: technologies integration, software components, distributed programming, and open implementations.

This report is based on the position statements of the authors, available at [OO WORKING GROUP 96], and on the discussions held during the ACM Workshop on Strategic Directions in Computing (MIT, June 1996).

1. WHAT IS OOP ABOUT ?

Object-oriented programming (OOP) is a technique for improving productivity, quality, and innovation in software development. Whereas procedures are sequences of statements that specify a transformation from inputs to outputs, objects are collections of operations sharing a state that provide persistent services over time. A procedure-oriented system consists of sequentially executed procedures, while an object-oriented system consists of a collection of interacting objects. Procedures express algorithms that implement computable functions, while objects express embedded systems that implement services over time, like banking or airline-reservation applications. Objects typically possess mutable state, operations that change their state, and operations that cause them to in turn invoke operations on other objects. Operations may be described through a well defined interface providing information about attributes and actions visible from clients. The implementation of each object is encapsulated – hidden from clients of the object.

Object-oriented programming is about modeling, reusability, and integration: it provides modeling primitives, a framework for high-level reusability, and integrating mechanisms for organizing knowledge about application domains.

—*Modeling*
OOP reflects the structure of the application domain by the structure of the program, because there is a direct one-to-one correspondence between objects of the application domain and objects of the computational model. Its concepts of class, subclass, inheritance, and virtual function allow structure at the level of primitive components to be extended to structure at the level of concepts, classification, and composition. If the application domain is the department of

[1]The contributors to this report are M. Aksit, A. Black, L. Cardelli, P. Cointe, J. Coplien, R. Guerraoui, G. Kiczales, D. Lea, O. Madsen, B. Magnusson, J. Meseguer, H. Moessenboeck, J. Palsberg, and D. Schmidt.

motor vehicles, the actual vehicles are represented by objects and the concepts of vehicle, car, bus etc are represented by classes of an inheritance hierarchy.

—*Reuse and extensibility*
Object-oriented languages (OOL) provide features for reusability and extension of software components. Given a class C, one may define an extension of C in the form of a subclass of C. The new subclass may redefine parts of C and/or add new attributes. This form of reuse and extensibility has turned out to be very useful in practice, especially for rapid development of prototypes. OOP also supports the extension of whole systems (frameworks) by allowing the programmer to plug objects of arbitrary subclasses into slots specified by abstract classes.

—*Integrating mechanisms*
The naturalness of its structuring mechanisms allows OOP to be used as an integrating framework for organizing knowledge about application domains over different phases of the software lifecycle. The structural correspondence between modeled objects and objects of the model is extended to a correspondence at the level of concepts by classes and inheritance, so that structures at the level of computing components correspond to structures of the application domain. Moreover, the same structure can be used for requirements, analysis, design, implementation, and maintenance phases of the software lifecycle. Though languages for different phases of the software life cycle may appear different, with analysis and design languages having a graphical syntax while programming languages have a textual syntax, they have the same language core. In addition to providing an integrated structure for phases of software development, object-oriented models also provide a natural framework for modeling databases and distributed systems. The robustness of OO systems in modeling application domains extends to the modeling of distributed systems, which may be regarded as simply another application domain for which object-orientedness (OO) is natural.

2. A BRIEF HISTORICAL PERSPECTIVE

OO started with programming. OOP originated with the Simula languages developed at the Norwegian Computing Center in Oslo, Norway [Birtwistle 73].

The first Simula language, Simula I, was originally created as a means for building simulation models. This required language support for modeling application domain phenomena and concepts. It was later realized that the principles of Simula I could be used for general-purpose programming. This lead to the development of Simula 67, a general programming language. Simulation was then supported by a set of classes called *Class Simulation*. Many of the concepts in OOP were available in Simula 67, including class, subclass, virtual function, and active objects in the form of coroutines. Class Simulation may be viewed as the first application framework.

The development of Smalltalk [Ingalls 76; Goldberg and Robson 83] at the Xerox Palo Alto Research Center in California further propelled OOP. Smalltalk is also based on the constructs class, subclass, and virtual function. However, unlike Simula, Smalltalk requires that *all* entities in a program be objects that can be manipulated via message sending. A consequence of this is that classes are also objects. Each class is an instance of another other class, called its metaclass. Another consequence is that expressions consist of sending messages to objects. An expression "3 + 4" is interpreted as sending the message "+" to the object "3" with argument "4". Smalltalk is not just a programming language. It is also a powerful programming environment with an excellent window based user interface. Smalltalk was developed together with graphical workstations and was the first system to demonstrate the power of workstations and graphical user interfaces. The simplicity, dynamics, and flexible nature of Smalltalk make it a powerful system for fast prototyping.

OOP experienced a rapid growth in industrial use with the advent of C++ [Stroustrup 86], a derivative of the C programming language. The use of C++ in large-scale development projects helped demonstrate that object-oriented constructs can be efficiently implemented and can support real-life applications.

3. STATE OF THE ART

Work in OOP may be grouped into the following categories, each illustrated by a few brief characterizations:

—*Foundations*
The term "foundations" means different things to different OOP researchers. For some, the principal issues surround support for modeling and development. Work in this area includes further development of tree-structured classification schemes (single inheritance) multiple classification schemes (multiple inheritance), and as dynamic classification hierarchies. Also, recent work on prototype based languages has led to the development of conceptual frameworks representing an alternative to the more traditional class-based view of modeling.
Other work in foundations includes more formal approaches to type theory, specification languages, object models, object-based concurrency, distributed object systems, and models of computation based on interacting objects. Each of these areas have been influenced by, and have in turn influenced, recent work in other areas of computer science.

—*Languages, environments, and tools*
Much of the work within OOP is concerned with language development. The major milestones for OOP were Simula, Smalltalk and C++. Recently Java has become a popular object-oriented language [Arnold and Gosling 96]. There are, however, a number of other object-oriented languages (OOL) with different strengths and focuses. The Smalltalk style of OOP quickly took on in the Lisp community and lead to Lisp extensions such as Flavors [Moon and Weinreb 80] and CLOS [Bobrow et al. 87]. Objective C [Cox 86] is a Smalltalk like extension of C. Eiffel [Meyer 89], BETA [Madsen et al. 93] are examples of statically typed languages. Prototype-based languages [Ungar and Smith 87] and actor languages [Smith and Hewitt 83], can to a certain extent, also be considered as object-oriented languages.
In addition to programming languages, there are a number of languages for analysis, design and data modeling. For example, the Unified Modeling Language integrates notations from several popular analysis and design methods.

—*Architecture and Design*
Research in extensibility has resulted in the notion of frameworks [Johnson 88]. A framework is a reusable design for an application or a part of an application

that is represented by a set of abstract classes, along with rules and conventions establishing how instances of these classes collaborate. A frameworks consists of a set of collaborating classes which make up a semi-finished system that is designed to be extended and adapted. Early examples are frameworks for graphical user interfaces where the framework defines a generic user interface with menus, windows and other user interface components but without any application-specific functionality. Such frameworks can be adapted by defining application-specific subclasses of menus and other components, and by plugging objects of these subclasses into designated slots of the framework. A common practice in OO software development is to focus on the design of the framework instead of the application. By doing this, it is often easier to modify or extend an application or reuse part of the code in other applications.

This grouping ((1) foundations, (2) languages, environments, and tools, (3) architecture and design) should not be taken as a strict classification of work in OO, since most work in OO has elements from all groups. For instance, in the development of a new programming language, the design may be based on well known concepts and constructs; it may be concerned with developing new language constructs from a technical programming point, without considering possible modeling aspects; it may primarily be concerned with extending a conceptual framework and supporting it by means of new language constructs; or it may be concerned with developing a new type theory and associated programming language constructs.

4. DIRECTIONS

While there are surely others, four specific areas representing major trends and/or open problems of OO stand out as central concerns over the next decade: technologies integration, software components, distributed programming, and the search for successors to the OO paradigm itself.

4.1 Technologies integration

To keep its place as a technique with a potential for industrial applications, OO technology has to support the demands for large-scale software development projects, e.g. efficiency, flexibility, safety, and standardization.

The first ACM OOPSLA conference (the main yearly US OO conference) was held in 1986. Given the attention the field has received, one would imagine that there would now be, some 10 years later, a wide range of systems to choose for industrial applications. Without going through the weaknesses of other languages and systems in detail, most people in industry would agree that there is only one language that has received general acceptance, C++. However, this language is constructed with enough compromises that its use has resulted in many projects whose successes and failures have at best an uncertain relation to OOP. While all kinds of other factors play a role, the expectation that "objects" would solve the software crises has not yet been fulfilled. In situations where C++ has been chosen as the only available alternative, it is easy to draw the conclusion that "objects do not work" or that "objects are just the same ideas in different clothing", although "mixed approaches get mixed results" is a fairer conclusion. This goes not only for language design, but also for other aspects such as training and design methods.

An important issue is to achieve industrial strength in object technology. To a large extent, this is more a question of getting existing solutions together in an integrated framework than opening up new areas of research. It is important to show how OO can be developed into an integrated common view on programming, as opposed to introducing elements of OO here and there in an existing framework. This integration approach has been the main focus of application of OO over the last decade, and it is time for a more coherent approach as a strategic direction for the next decade.

A consistent combination of at least the following issues should be provided:

—Computational model;
—Language design: completeness in functionality, simplicity, and readability;
—Language implementation;
—Efficiency with scaling-up to large systems (without over-night compilations);
—Compatibility of design notations and modeling capabilities with languages;
—Prototyping of the development process;
—Environments.

Each of these topics can be the focus of research in its own right, but OO has focused on such a level of "tactics" for too long and now needs more work on a "strategic" level where the focus is to find coherent solutions through all (or several) of these aspects.

4.2 Software Components - Patterns and Frameworks

The state of affairs in most organizations today is commonly referred to as the "software crisis". Over the past decade, hardware has become increasingly smaller, faster, and cheaper, whereas software has become larger, slower, and more expensive to build and maintain. Much of the cost and effort associated with building software go into rediscovering fundamental software concepts and reinventing common software components.

Design patterns are a promising approach for alleviating the costly rediscovery of fundamental software concepts and abstractions. A design pattern is a recurring solution to a standard software development problem. Design patterns help developers leverage the experience of others by (1) communicating knowledge of successful software architectures, (2) making it easier to incorporate a new design paradigm or architectural style, and (3) avoiding traps and pitfalls that have traditionally been learned only by trial and error [Coplien and Schmidt 96].

Patterns are particularly useful for documenting software architectures and design abstractions that have proven useful over time. However, these abstractions do not directly yield reusable code. Therefore, it is essential to augment the study of patterns with the creation and use of *frameworks*. Frameworks help developers avoid costly reinvention of common software components by providing "semi-complete" application skeletons that can be customized by inheriting and instantiating from reusable building blocks. Since frameworks are tightly integrated with a particular domain (such as user interfaces, telecommunication switching, or avionics), the scope of reuse can be significantly larger than by using conventional class libraries.

Over the past decade, extensible frameworks (such as MacApp, ET++, Interviews, Choices, MFC, implementations of OMG CORBA and Microsoft DCOM, Java's AWT, etc.) have played a large role in shaping contemporary software architectures [Gamma et al. 95]. More recently, a body of literature on design patterns has emerged [Coplien and Schmidt 96; Coplien et al. 96; Buschmann et al. 96]. These patterns identify, document, and catalog successful solutions to common software problems. The patterns captured by this literature have already had a significant impact on the construction of commercial software [Schmidt 96; Beck et al. 96]. In these systems, patterns have been used to enable widespread reuse of communication software architectures, developer expertise, and object-oriented framework components.

Over the next few years, we anticipate that a wealth of software design knowledge will be captured in the form of patterns and frameworks. These patterns and frameworks will span domains and disciplines such as concurrency, distribution, organizational design, software reuse, real-time systems, business and electronic commerce, and human interface design. We expect the following aspects of patterns and frameworks will receive particular attention [Schmidt 95]:

—*Integration of design patterns together with frameworks*
Framework developers are confronted with many challenging design tradeoffs. One of the most crucial is determining which components in a framework should be variable and which should be stable [Pree 94]. Insufficient variation makes it hard for users to customize framework components. This results in a framework that cannot accommodate the requirements of diverse applications. Conversely, insufficient stability makes it hard for users to comprehend the framework. Inflexibility and instability can create a framework that is awkward to use and unable to satisfy other requirements (such as run-time performance).
Many design patterns are intended to decouple the stable portions of software from the variable portions [Gamma et al. 95]. Such patterns can be viewed as abstract descriptions of simple frameworks that facilitate widespread reuse of software architectures. Similarly, frameworks can be viewed as concrete realizations of patterns that facilitate direct reuse of design and code.
One difference between patterns and frameworks is that patterns are described in language-independent manner, whereas frameworks are generally implemented in a particular language. However, patterns and frameworks are highly synergistic concepts, with neither subordinate to the other. We expect that the next generation of object-oriented frameworks will explicitly embody many patterns, and patterns will be widely used to document the form and contents of frameworks.

—*Integration of design patterns to form pattern languages*
Much of the existing literature on patterns is organized as catalogs of design patterns [Pree 94; Buschmann et al. 96]. These catalogs present a collection of relatively independent solutions to common design problems. As more experience is gained using these patterns, developers and authors will increasingly integrate groups of related patterns to form *pattern languages*. These pattern languages will weave together a family of patterns that document more complete solutions to both "development-centric" domains (such as real-time systems, business applications, and electronic commerce) and "human-centric" domains (such

as organizational structure and human interface design).

Just like frameworks support larger-scale reuse of design and code than do standalone functions and class libraries, so will pattern languages support larger-scale reuse of software architecture and design than do individual patterns. Developing pattern languages is challenging and time consuming, but we believe they will ultimately provide the greatest payoff for developing high quality software.

—*Integration with current software development methods and software process models*

Patterns help to alleviate software complexity at several phases in the software lifecycle. For instance, patterns can help developers navigate through alternative choices *within* a particular development phase. In the analysis and design phases, for example, patterns can help guide developers in selecting from software architectures that have proven to be successful. Likewise, in the implementation and maintenance phases, patterns help document the strategic properties of software systems at a level higher than source code.

In addition, patterns can help developers navigate abstraction boundaries *across* software development phases. For instance, patterns help to bridge the abstractions in the upstream phases (such as domain analysis and architectural design) with the concrete realizations of these abstractions in downstream phases (such as implementation and maintenance).

The patterns and pattern languages that exist today do not yet form a comprehensive software development method or complete process guide. However, they do complement existing approaches by focusing on non-functional forces (such as backwards compatibility or architectural extensibility) that are often not addressed by conventional development methods and processes.

4.3 Distributed Programming

It is increasingly the case that all interesting computing systems are distributed. As soon as a system has multiple users, we find that they wish to collaborate – to share objects with each other – using a variety of computers in a multiplicity of locations.

It has been argued that the World Wide Web is itself a distributed object-oriented system [Black 94]. The Web provides uniform object names (URLs), persistent object storage, and delivery of invocation messages to objects. It is a rather poor implementation of an object system, in that objects cannot move without changing their identity (their URL); the number of types of objects is small; and creating new types of objects requires a global agreement in the form of a new version of the http protocol or the introduction of new protocols. Nevertheless, the Web is probably the most widely used object system in the world; it is manifestly useful, and multiple implementations of all of the important pieces are in common use.

Over the next ten years we will be building object systems that improve on the capabilities offered by the Web today. The last vestiges of centralized control will disappear; they are too great a hindrance to composition and the autonomous evolution of components. Language features such as class variables, *a priori* declaration of subtyping relationships, and centrally dispatched multimethods cannot easily be used in the distributed environment.

Distribution will also convince us to take encapsulation seriously. While encapsulation is the bedrock on which objects are built, programmers are still prone to give up encapsulation to gain other properties, such as efficiency. When distributed programs cross administrative boundaries, this is not an option. Distribution is also a constant reminder that *Type* (the interface of an object) and *Class* (its implementation) are distinct concepts. If one company is selling a service over the network in the form of an invocable object (such as a web server), it is important to the company that (1) customers cannot look at the implementation and (2) the service that their object offers is functionally indistinguishable from that offered by a competitor, even though the two companies use independently developed implementations.

Large distributed systems grow by evolution (the gradual inclusion of new objects with new capabilities) rather than by revolution, where the old system is shutdown and a new, improved system is substituted for it. This means that they must support separate compilation (separate in space as well as separate in time). The compiler and run-time system must not need global knowledge to do its work. New classes may be added to a running system freely.

New types can also be added to a running system. Subtyping (conformity) tells programmers and users when a new object can be treated as an enhancement of an existing object. It formalizes what protocol designers have understood in an *ad hoc* fashion for many years: the ways in which a protocol can be changed while retaining backward compatibility for old code.

The location of objects in a distributed system has massive impact on performance. Invoking a remote object rather than a local one may not change the semantics of the application, but it may degrade the performance by three orders of magnitude. An operation that might complete locally in under a second might thus take over a quarter of an hour. Confronted with such a system, most customers would believe that it no longer works.

Distributed object-oriented systems must therefore include mechanisms for moving an object to improve performance, without changing the semantics or the name of the object concerned. Initially, such movement will be under programmer control, but we will need to devise ways in which the placement of objects can be controlled automatically by the system itself, much in the same way that persistent objects migrate from disk to main memory and back again without direct programmer intervention. Quality-of-service abstractions will be developed so that the client's expectations and the server's guarantees can be codified, and the system can adapt to those requirements.

It will also be necessary to integrate security into programming languages and mobile computations. As compilation ceases to be a once-in-a (program's) lifetime occurrence, and becomes a routine part of execution, we must integrate type checking and encapsulation boundary enforcement.

Although large scale distributed systems will probably be written in a multiplicity of programming languages, it is essential that all these languages understand a common notion of object interface and object invocation. An object type description language would codify that understanding and play the same role in distributed OO programs as an RPC interface description language does in conventional distributed systems.

A major challenge will be reconciling the need for semantic uniformity (which implies that all objects must be treated identically regardless of their location) with the need to build robust distributed systems. The latter implies that failures must be contained, and that dependencies of one component of the system on another (remote) component must be minimized. Remote reference to a single (remote) object preserves uniformity, but sacrifices robustness. Copying, replication, and caching increase robustness, but sacrifice the semantic simplicity of a uniform object model. In current object-oriented languages, a replicated entity cannot even be treated as a (single) object.

4.4 OO inventing its successor

The OO paradigm has proven to be extremely powerful and widely applicable. But experience pushing the limits of OO's ability to deliver highly reusable, tailorable and flexible code has shown that the paradigm has its limitations, and has caused a number of researchers to reexamine OO's basic promise.

In simple terms, that basic promise is that by modeling everything in terms of the object paradigm (objects, classes, instance variables etc.), OO will give the programmers tremendous flexibility ... And this is true in cases where it works.

But this pure object model turns out to be more brittle than one would like. This brittleness leads to a number of difficulties, in all phases of the program lifecycle. What follows is an overview of some of these problems, and some of the efforts that are already underway to address those limitations.

What containment structure?. One problem caused by the overly rigid nature of the object model is the pervasive difficulty in settling on the object containment structure for a system. As an example, consider an object model of an automobile. One might choose at first to have a *car* object that contains (points to) four *wheel* objects. But then, in a later stage of development one might want to have the *car* object contain a *drivetrain*, which itself contains four *wheels*. This may appear to be a trivial change, but in the OO paradigm, making this change to a large system can be surprisingly uncomfortable.

One approach to addressing this kind of problem is adaptive programming (AP) [Lieberherr 96]. AP works by splitting the OO program into two parts: the behavior and the object containment structure (class graph). A special language mechanism, called a succinct subgraph specification allows the behavior part of the program to automatically adapt to a large number of changes in the class graph. This effectively loosens the object model enough that makes many maintenance and evolution tasks for OO programs become simpler when using AP. Similar problems arise having to do with what to model as a class versus an object, and what to model in the state (instance variables) of an object versus in its class. Again there are important differences, and changing a program from one to another is surprisingly expensive.

Solutions that have been proposed for these problems include prototype-based languages, in which there is no distinction between classes, normal objects, and predicate classes, which attempt to blur the distinction between changing an object's state and changing its class.

Composing objects. Another set of problems arise in composing sets of objects, when an issue arises that doesn't align with the OO model. A well known example of this is the "inheritance anomaly" [Matsuoka and Yonezawa 93]. The problem in this case is that an object's behavior and its synchronization strategy simply don't both fit into the same inheritance mechanism. There are similar problems having to do with transactions, which do not align well at object boundaries [Guerraoui 95]. The work on composition filters aims at making object-oriented programs more adaptable and reusable through modular and orthogonal extensions [Bergmans and Aksit. 96]. A modular extension means that one or more composition filters can be attached to an object, for example expressed in C++ or Smalltalk, without modifying the object's basic structure. Each filter may address a particular concern, such as multiple interfaces, dynamic inheritance and delegation, inter-object coordination, synchronization and real-time constraints. Orthogonality of filters provides composition of the object's basic behavior with these additional concerns without causing inheritance anomalies.

Reflection. In general, reflection in OO languages provides the programmatic ability to step outside of a particular object and examine and change some aspect of the execution environment [Kiczales 91]. This works by providing a meta-level handle on the program and the computing context. For example, a reflective OO language can be used to provide access to the policies and mechanisms of message sending in a distributed OO system.

Adaptive Programming can also been seen in these terms. It provides a meta-level handle on the class graph, makes it possible to "see farther" than an object's immediate children, but to do so in a principled way (i.e. without violating encapsulation).

Open implementation [Kiczales 96] is an approach to open implementations by providing them with a second interface which allows to control implementation policies. This leads to more flexible software than closed or black-box implementations. Reflection is often used to implement open implementation.

Aspect-Oriented Programming. What all of these efforts appear to have in common is the development of a separate linguistic mechanism for addressing some aspect of the system that is difficult to capture in the pure object model. Adaptive programming adds graph language mechanisms, to add flexibility in object containment structure. Composition filters add a language for controlling the message-based interaction between objects, to provide a handle on issues such as synchronization. Reflective mechanisms add more general-purpose meta-languages for controlling the behavior of the object language itself.

More recently, some researchers have begun to generalize this approach, under the name of aspect-oriented programming (AOP) [Aspect Oriented Programming 96]. In AOP, the primary functionality of a system is written using whatever paradigm is most appropriate (i.e. OO), and then those aspects of the system that do not fit naturally into that paradigm are written using more special purpose languages. The resulting set of programs in different languages are combined by a special kind of compiler called a weaver.

As an example consider a complex distributed information system. Using AOP, the main functionality might be programmed using an OO language, while the replication, communication and distribution might each be programmed in their own special-purpose language. The result would then be automatically woven together to produce executable (for example, C) code.

5. CONCLUSION

This paper has provided an overview of the field of object-oriented programming. After presenting a historical perspective and some major achievements in the field, four research directions were introduced.

In general there is a need to continue research in traditional areas: (1) As the computer systems become more and more complex, there is a need to further develop the work on architecture and design; (2) to support the development of complex systems there is a need for better languages, environments and tools; and (3) foundations in the form of the conceptual framework and other theories must be extended to enhance the means for modeling and formal analysis, as well as for understanding future computer systems.

REFERENCES

M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object-Interactions Using Composition-Filters. In R. Guerraoui, O. Nierstrasz and M. Riveill, editors, *Object-Based Distributed Programming*, Springer Verlag, LNCS 791, 1993.

AOP Home page: "http://www.parc.xerox.com/aop".

K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series, Addison-Wesley, 1996.

K. Beck, R. Crocker, J. Coplien, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides. Industrial Experience with Design Patterns. *International Conference on Software Engineering*. March 1996.

G. Birtwistle, O. Dahl, B. Myhrhaug and K. Nygaard. *Simula begin*. Petrocelli Charter, New York, 1973.

A. Black. Objects to the Rescue. In *ACM SIGOPS European Workshop*, 1994.

L. Bergmans and M. Aksit. Composing Synchronization and Real-Time Constraints. *Journal of Parallel and Distributed Computing*, Vol. 39, No. 1, pages 32-52, July 1996.

D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales and D. Moon. *Common Lisp Object System Specification*. Technical Report 87-002, ANSI Common Lisp Committee X13J13, 1987.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

J. O. Coplien and D. C. Schmidt, eds. *Pattern Languages of Program Design*, Vol 1. Addison-Wesley, 1995.

J. O. Coplien, J. Vlissides, and N. Kerth, eds. *Pattern Languages of Program Design*. Addison-Wesley, Vol 2. Reading, MA, 1996.

B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

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

R. Guerraoui, Modular Atomic Objects. *Theory and Practice of Object Systems*, Vol. 1, n° 2, John Wiley and Sons, November 1995.

A. Goldberg and D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading, Mass, 1983.

B. Smith and C. Hewitt. *A Plasma Primer*. AI Laboratory, MIT, September 24, 1975.

D. Ingalls. The SMALLTALK-76 Programming System Design Implementation. *ACM Conference on Principles of Programming Languages*. Tucson, Arizona, 1978.

R. JOHNSON and B. FOOTE. Designing Reusable Classes. Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35.

G. KICZALES, J. DES RIVIÈRES, and D. BOBROW, *The Art of The Meta-Object Protocol*, MIT-Press, 1991.

GREGOR KICZALES. Beyond the Black Box: Open Implementation. *IEEE Software*, January, 1996.

K.J. LIEBERHERR, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

O.L. MADSEN, B. MOLLER-PEDERSEN, and K. NYGAARD. *Object Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993.

S. MATSUOKA and A. YONEZAWA. Analysis of inheritance anomaly in object-oriented concurrent programming language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming,* pages 107-150. MIT Press, 1993.

B. MEYER. *Eiffel: The Language, version 2.2.* Interactive Software Engineering, Inc., Santa Barbara, CA, USA, 1989.

D. MOON and D. WEINREB. *Flavors: Message Passing in the LISP Machine.* AI Memo 602, AI Lab, MIT, Cambridge, Mass, 1980.

Position statements of the participants in the Object Oriented Programming Working of the *ACM Workshop on Strategic Directions in Computing.* "http://lsewww.epfl.ch/ rachid/conferences/oowg/oowg.html".

W. PREE. *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, MA, 1994.

D. SCHMIDT. Using Design Patterns to Develop Reusable Object-Oriented Communication Software, *Communications of the ACM*, (Special Issue on Object-Oriented Experiences, Mohamed Fayad and W.T. Tsai Eds.), 38, 10, October 1995.

D. SCHMIDT. A Family of Design Patterns for Application-Level Gateways. *Theory and Practice of Object Systems*, John Wiley and Sons, to appear 1996.

B. STROUSTRUP. *The C++ Programming Language.* Addison-Wesley, 1986.

D. UNGAR and R. SMITH. Self: The power of Simplicity. *ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'96).* 22(12), pp 227-242, 1987.