Object Buyer's Guide p.25 • Coding and Design with JavaBeans p.14

The Global Authority on Object Development

The Journal of Object-Oriented Programming
January 1998 Vol. 10, No. 8

Maximizing Your Database Performance

Maintaining OO Legacy Systems by Recombining Components

Reducing Deadlocks and Increasing Concurrency

Using RC Objects to Improve Concurrency

ANDREW KOENIG
What the C++ Standard Will Do for Us Now

LALONDE & PUGH
Building a Voice-Controlled Browser

BERTRAND MEYER
Combining Components with Eiffel

US $9/Can $12/Intl $13

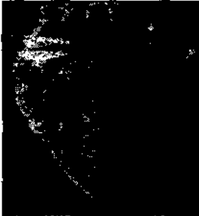
Get Ready for Object Expo/SIGS Expo for Java™ p. 1

EDITORIAL

Happy New Year! Welcome to the first issue of 1998. *JOOP* initiates its 1998 publishing year with an unusually large collection of papers, columns, and reviews.

I would like to welcome Bertrand Meyer to our roster of columnists. His new column will focus on OO software engineering and Eiffel. I believe that you will find his first column, "The Component Combinator for the Enterprise," very interesting.

I would like to welcome back our regular columnists, Andrew Koenig (C++), Kumar Vadaparty (ODBMS), John McGregor (Quality Assurance), Desmond D'Souza (Modeling and Design with Java), and Wilf LaLonde and John Pugh (Smalltalk). They will be joined by other regular columnists as the publishing year unfolds. Limited editorial space does not allow all of our columnists to be published at the same time.



Dr. Richard Wiener
EDITOR

We have seven feature-length articles in this issue of *JOOP*: five *JOOP* articles and two *ROAD* articles. As is fairly common, our authors hail from all parts of the world (as do our readers). Eduardo Casais ("Re-Engineering Object-Oriented Legacy Systems") is from Nokia Research Center in Finland. E. Marcos, A. De Miguel, and M. Piattini ("About Abstract Classes") are from the Departmento de Informatica in Madrid. Ari Jaaksi ("A Method for Your First Object-Oriented Project") is also from Nokia in Finland. Ted Foster is from Class Software Construction (UK) and Liping Zhao is from the Department of Computer Science, RMIT, Australia ("Modeling Transport Objects with Patterns"). Woochun Jun and Le Gruenwald ("Semantic-Based Concurrency Control in Object-Oriented Databases") are from the Department of Computer Science, University of Oklahoma. Jay Almarode and Robert Bretl ("Reduced-Conflict Objects") are from GemStone Systems in Beaverton, OR. Finally, David Gillibrand and Kecheng Liu ("Quality Metrics for Object-Oriented Design") are from the School of Computing at Staffordshire University in the UK. Quite an international lineup!

As is customary in the first issue of each year, I would like to extend an invitation to all of our readers across the world to send their papers for publication in *JOOP*. These papers should be aimed at an audience best characterized as intermediate to advanced software engineers and programmers. The papers can focus on any aspect of object technology. The review process shall be prompt. Please send a cover letter including contact information and email address, three hard copies of the paper, and a floppy disk containing the word processing document to me at: Richard Wiener, Editor *JOOP*, 135 Rugely Ct., Colorado Springs, CO 80906. You can contact me quickly at rswiener@elbert.uccs.edu. Please do not send your manuscripts via e-mail.

Let's have a great year!

Richard S. Wiener

JOOP (ISSN #0896-8438) is published nine times per year, monthly except combined Mar/Apr, Jul/Aug, and Nov/Dec by SIGS Publications Inc., 71 West 23rd Street, 3rd floor, New York, NY 10010. Please direct advertising inquiries to this address. Periodicals postage paid at New York, New York, and additional mailing offices. POSTMASTER: Send domestic address changes and subscription orders to JOOP, P.O. Box 5050, Brentwood, TN 37024-5050. Inquiries and new subscription orders should also be sent to that address. Annual subscription rates for the US are $199 for institutions, $79 for individuals. All foreign orders must be prepaid in US funds drawn on a US bank. Canadian & Mexican orders add $25 per year and non-North American orders add $49 per air service. For service on current domestic subscriptions, call 800.361.1279, fax 615.370.4845, e-mail subscriptions@sigs.com. For foreign subscriptions and inquiries Phone: +44(0)1858 435302.

© Copyright 1998 SIGS Publications Inc. All rights reserved. Reproduction of this material by electronic transmission, Xerox, or any other method will be treated as a willful violation of the US Copyright law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Statements of opinion and fact are made on the responsibility of the authors alone and do not imply an opinion on the part of SIGS PUBLICATIONS INC. or the editorial staff. All trademarks are the property of their respective owners.

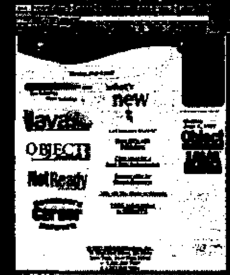
Manuscripts under review should be typed double spaced (in triplicate) and accompanied by an electronic file in TEXT format. Editorial correspondence and Product News information should be sent to the Editor, Dr. Richard S. Wiener, 135 Rugely Court, Colorado Springs, CO 80906, 719.579.9616 (voice & fax).
Printed in the USA. Canada Post International Publications Mail Product Sales Agreement No. 290343.

JOOP

The Journal of Object-Oriented Programming

The Global Authority on Object Development

January 1998 Vol. 10, No. 8



Editorial 4

Eiffel 5
The Component Combinator [illegible] Enterprise Applica[illegible]
Betrand Meyer

C++ 10
A Quiet Revolution [illegible]
Andrew Koenig

Modeling & Design with Java [illegible] 14
JavaBeans: Coding [illegible] Design
Desmond D'Souza

Quality Assurance 60
Building Tests From Specifications [illegible]
John D. McGregor

ODBMS 65
A Closer Look at [illegible]
Kumar Vadapa[illegible]

Smalltalk 69
Building A Speech-[illegible] Recipe Browser
Wilf LaLonde [illegible]

Book Review 76
Use Cases Combined with Booch/OMT/UML: Process [illegible] Products
Reviewed by Ian Graham

Correction 25

Ad Index 64

OOP University 77

Product News 78

Recruitment 80

Cover: Steven Hunt

A Method for Your First Object-Oriented Project 17
Ari Jaaksi
Object-oriented literature provides a plethora of methods and notations, many of which are complicated and difficult to learn and use. Ari Jaaksi discusses a pragmatic and simple approach—just two notations and five steps—toward finding a method that is right for your project.

Modeling Transport Objects with Patterns 26
Ted Foster and Liping Zhao
Design patterns have provided a way to describe frequently occurring components within public transport object models. They have also delivered solutions for previously difficult problems. The Role (State) and Strategy patterns are used to model transport objects that play multiple concurrent roles and contain multiple solutions.

Semantic-Based Concurrency Control in Object-Oriented Databases 33
Woochun Jun and Le Gruenwald
Supporting concurrency control in OODBs is more complex than it is in its relational equivalents. To overcome this problem, runtime information is used to increase concurrency. The authors look at semantics of methods, nested method invocation, and referentially shared objects, and show how to automate communicativity of methods.

Reduced-Conflict Objects 40
Jay Almarode and Robert Bretl
By carefully defining semantics, implementors can prevent concurrency conflicts for particular sequences of operations. While locking solves part of the problem, it is also an expensive solution. Object databases provide better alternatives because they allow a higher level of semantic manipulation; operations on objects can be more sophisticated than simple read and writes.

Re-Engineering Object-Oriented Legacy Systems 45
Eduardo Casais
A drawback to the rising popularity of object-oriented technology is the indiscriminate use of OO mechanisms, and weak analysis and design methods. This leads to large legacy systems that are inflexible and hard to maintain. Eduardo Casais examines OO re-engineering experiences and identifies ways to better maintain legacy systems.

About Abstract Classes 53
E. Marcos, A. de Miguel, and M. Piattini

Quality Metrics for Object-Oriented Design 56
D. Gillibrand and K. Liu

By Esperanza Marcos, Adaración de Miguel, and Mario Piattini

About Abstract Classes

"...the level of abstraction that we can speak directly affects the size of the problem we can solve."[6]

Object-orientation has increased the level of abstraction with respect to previous paradigms, appearing as a powerful mechanism of knowledge representation that makes the resolution of complex problems easier. In addition, a higher level of abstraction facilitates software reuse, extending this concept beyond the traditional to admit other types, such as knowledge reuse.

Two concepts that contribute substantially to increase the level of abstraction and reuse arise in the scope of object-orientation: On one hand is the concept of classes that do not implement all their methods, but delegate this task to their subclasses; on the other is the concept of noninstantiable classes. To refer to these two concepts, two terms appear in object terminology: *deferred class* and *abstract class*. However, there is some confusion between these two terms, and also between the concepts that they represent: On many occasions these terms are considered synonyms, and the concepts they represent, equivalent; some authors, though, only speak of abstract classes, others only of deferred classes, etc. We strive to contribute to the clarification of this controversy, giving a separate meaning to each of these terms, and establishing the difference that exists between the concepts they represent.

ABSTRACT AND DEFERRED CLASSES IN THE LITERATURE

To illustrate the controversy we referred to in the previous paragraphs, we will give examples of some of the definitions that can be found in the literature.

The concept of deferred class was introduced by Meyer,[7] who defined deferred class as "a class that contains deferred routine." An immediate consequence of this definition is that a deferred class cannot be instantiated directly, but only through its subclasses. Meyer[7] posed the following deferred class noninstantiation rule: "Create may not be applied to an entity whose type is given by a deferred class."

The concept of deferred class has been extended in the object paradigm, but some authors refer to it under the name of "abstract class." This is the case in C++. In the context of C++, Bjarne Stroustrup writes, "a class with one or more pure virtual functions is an abstract class, and no objects of that class can be created."[10]

However, other authors give different definitions for the concept of abstract class. For Wirfs-Brock *et al.* "classes that are not intended to produce instances of themselves are called abstract classes."[12] This is also the case with the ODMG-93 model,[2] and with STEP/EXPRESS,[9] where an abstract class is one that can only be instantiated through its subclasses.

Another variant in the use of these two terms is the one employed in the Unified Method (UM), where a deferred class is a noninstantiable class: "deferred class means that it must be subclassed and may not be instantiated (also known as an abstract class, but we use Meyer's term here because it is more descriptive)."[1] The UM considers abstract class and deferred class concepts equal, referring to both concepts as deferred classes.

BUT ARE ABSTRACT CLASSES AND DEFERRED CLASSES THE SAME THING?

If we rest on the definitions that can be found in object terminology, it seems that there is no uniform criteria. In UM there is no difference between the two concepts, and the definition of deferred class as a class that may not be instantiated is attributed to Meyer. Nevertheless, in our opinion, the concept of deferred class in the UM (noninstantiable class) is a direct consequence of the definition given by Meyer for this same term (a class that delegates the implementation of some of its methods to its subclasses).[7] Therefore we cannot say that the concepts of deferred class in the UM and in Meyer[7] are strictly equivalent.

We believe that there is a substantial difference between the two terms in discussion. It is true that abstract and deferred classes are noninstantiable (and so, both have to be subclassed); however,

E. Marcos, A. de Miguel, and M. Piattini work in the Departamento de Informatica, at the Universidad Carlos III de Madrid. They can by reached by email at {cuca, mpiattin, admiguel}@inf.uc3m.es.

it is not true that every noninstantiable class defers, necessarily, the implementation of any of its methods to its subclasses. An abstract class could (or could not) delegate the implementation of any methods to its subclasses; if this happens, according to Meyer, we would be referring to deferred classes. Both types of classes, abstract and deferred, are instantiable only through their subclasses.

Wirfs-Brock *et al.*[12] only speak of abstract classes, but they make the same distinction that we propose: "Abstract classes can be designed in two distinct ways: They can provide fully functional implementations of the behavior which they exist to capture; or they can provide a template for behavior that is intended to be defined by specific subclass methods."

The concept of abstract class has a higher level of abstraction (as its name suggests) than the concept of noninstantiable class.

Hürsch[5] also proposes a distinction between the two concepts: the ability to create instances and the presence of deferred methods (he calls them abstract methods). He defines the abstractness of a class as "the inability to instantiate objects" and he remarks, "note that this definition deliberately does not make any reference to, and thus is independent of, the presence of abstract methods."* According to Hürsch[5] "if a class contains abstract methods, then the class cannot be safely instantiated, because, if an instance were created, it would not be able to respond successfully to all of its messages. So the presence of an abstract method implies the inability to instantiate objects. However, the converse does not hold: a class might not be intended for instantiation and yet might have no abstract methods." Hürsch[5] also proposes adding a constructor to the object-oriented languages that would allow the definition of abstract classes to be independent from the definition of deferred methods, just as it has been until now. In Eiffel, like in Smalltalk or in C++, a class is abstract if it has some deferred method.†

In general, we agree with Hürsch's approach,[5] but we make a finer distinction based on the difference between abstract classes and noninstantiable classes. In our opinion, the concept of abstract class has a higher level of abstraction (as its name suggests) than the concept of noninstantiable class. We think the inability to instantiate objects is a direct consequence of the abstract-class concept, rather than its definition. In accordance with this, we can define an *abstract class* as "a class that has no direct correspondence with a description of any entity distinctly identified in the universe of discourse and which is introduced in the model to increase the level of abstraction."

The distinction between abstract and deferred classes is not only interesting conceptually, but also concerns the stage of development in which each of these two concepts is introduced: The notion of abstract class corresponds to the analysis phase, because it is introduced to increase the level of abstraction. The deferred class is a design concept, because it is in this phase that the different possibilities of implementation appear, and its aim is reuse. Generally speaking, we can say that the superclass will be abstract in every hierarchy where we come through generalization (regardless of whether it implements its methods or not) whereas, in the cases where it comes through specialization, the superclass will be abstract only when the hierarchy will be total (the union of the subclasses covers the superclass as a whole). An abstract class, such as the one Wirfs-Brock *et al.* propose,[12] may or may not be designed as deferred. Because each abstract and deferred class is noninstantiable, they must be subclassed.

According to this, in those models that distinguish between type and class (where the class is the implementation of the type), we would speak of abstract types in the analysis phase, and abstract or deferred classes in the design and implementation phases.

IS AN INTERFACE AN ABSTRACT CLASS?

The problem of abstract classes is considered nowadays, as Wegner states,[11] as a representative open research issue; its application is not restricted to the inheritance hierarchies, but can be applied to distributed systems where the importance of autonomous interfaces is a main requirement for improving interoperability: "Separation of behavior from instances is a central interoperability requirement of interface definitions languages of ODMG's CORBA and Microsoft's DOM. Abstract interfaces have stronger abstraction needs than abstract inheritance classes, since they require all their method implementations to be virtual (deferred), but both separate interface specification from instance creation."

Wegner's[11] thought is based on an idea proposed by Hürsch,[5] who distinguishes between interface inheritance and data inheritance. However, the meaning for abstract class that Hürsch now proposes is different from those proposed previously; he states that: "interface inheritance occurs when the only thing a class inherits is a set of interfaces defined in the superclass. In particular, the superclass does not provide any implementation, nor does it define any data representation, which effectively makes it an abstract superclass."

We can infer that Hürsch[5] considers every interface to be an abstract class, because all their methods must be deferred. However, in our opinion, the meaning of deferred method here is not the same as the meaning of deferred method in Meyer's terminology. The implementation of an interface is not the responsibility of its subinterfaces, the responsibility belongs to the class that implements the interface. According to the distinction just mentioned between type and class (as the implementation of a type), we can say that the interface is the type (therefore, the interface does not have to implement its methods) and the class is its implementation. This is the case with some object models, like the ODMG-93 model and the OMG model,[13] where a main

* Hürsch's "abstract method" is Meyer's "deferred routine."
† Abstract method in Smalltalk, pure virtual method in C++, and deferred method in Eiffel.

objective is, precisely, to provide support to distributed objects. So, for example, in the OMG object model, types categorize objects that have the same interface, and classes categorize objects that have the same implementation.

Hürsch[5] also poses another interesting problem that arises from the notion of the class as a factory, so, therefore every class must provide a service to create instances. This notion of class poses a problem because neither abstract, nor deferred classes can have instances by themselves, so these kinds of classes will never be able to achieve the functions of a factory.

ARE THERE OTHER NONINSTANTIABLE CLASSES?

According to what has been stated here, we can assume that there are at least two kinds of noninstantiable classes: abstract classes and deferred classes. But now we can pose the following question: Could we find a noninstantiable class that is neither abstract nor deferred? At least four kinds of noninstantiable classes appear in the literature, each one of them answering to different needs. Some, but not all, of them are abstract:

- The first case, as we have already discussed, is abstract classes. This kind of noninstantiable class appears in the analysis phase because of abstraction reasons, usually classification abstraction, and implies software reuse.
- The second case, also discussed, is deferred classes. These are introduced in the design phase to increase the software's reusability. A deferred class can come from an abstract class in the analysis phase, although an abstract class must not always be designed as a deferred class.
- Although it may seem that every noninstantiable (abstract or deferred) class is used because of abstraction and reuse needs, there are some cases where this point is not so clear. Consider the case of parallel inheritance which appears in the model of the UM, among others. In Figure 1 we show an example taken from the UM.

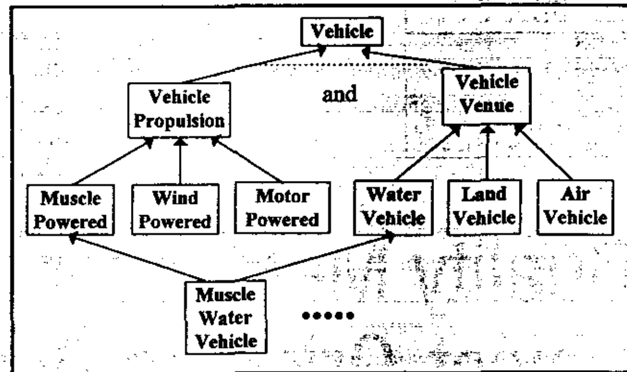


Figure 1. Example of parallel heritage in the UM.

In this example, Muscle, Wind, and Motor Powered, as well as Water, Land, and Air Vehicle, are noninstantiable classes. However, this restriction is not due to the fact that such classes have been introduced, as in the case of abstract classes, because of the abstraction reasons, but because of the model's needs, since it is the way that UM supports overlapping.‡

Nevertheless, even though the ultimate reasons that noninstantiable classes are introduced in a hierarchy of parallel heritage are not the same ones by which abstract classes are introduced in a model, the final effect is similar: a classification hierarchy in which noninstantiable classes are an abstraction with regard to their subclasses. For this reason, perhaps this kind of noninstantiable class could also be considered as abstract class.

- Some object models require each family of types to have a single maximal supertype. This restriction forces us, as Melton[8] states, to invent a noninstantiable type that will be used as a maximal supertype. This is, for example, the case with the Denotable_Object type in the types hierarchy of the ODMG-93 model. This case corresponds to the case of a noninstantiable class, although it would be debatable whether or not it is an abstract class. Actually, this refers to a generic class with regard to its subclasses, but it is introduced in an artificial way. Because of this, the subclasses will probably not have any common characteristic that could be inherited from this new superclass.

Despite the fact that this is the only example of noninstantiable and nonabstract (or deferred) class that we have found, there could be others. In such a case, perhaps we would have to distinguish between noninstantiable, abstract, and deferred classes.

CONCLUSION

In this paper we have tried to clarify, as far as possible, the confusion regarding two very important concepts in the object-oriented paradigm: abstract and deferred class concepts. Some authors use these two terms to refer to the same concept: a class that delegates the implementation of some of its methods to their subclasses. Other authors also use both terms as synonyms, but with a different meaning from the one above: classes that cannot be instantiated by themselves. Finally, other authors only speak of abstract classes as those that cannot be directly instantiated.

We establish the difference between such concepts; defining deferred class according to Meyer,[7] and abstract class as "a class that has no direct correspondence with a description of any entity distinctly identified in the universe of discourse, and that is introduced in the model to increase the level of abstraction." Furthermore, while abstract classes are introduced in the analysis

continued on page 59

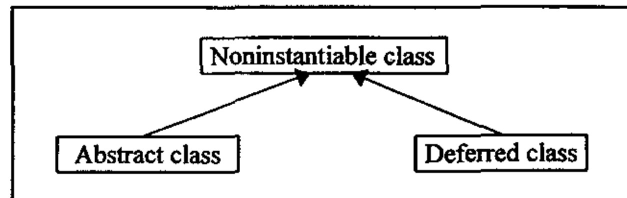


Figure 2. Generalization of noninstantiable classes.

‡ This restriction was removed from UML,[14] because UML directly supports overlapping.

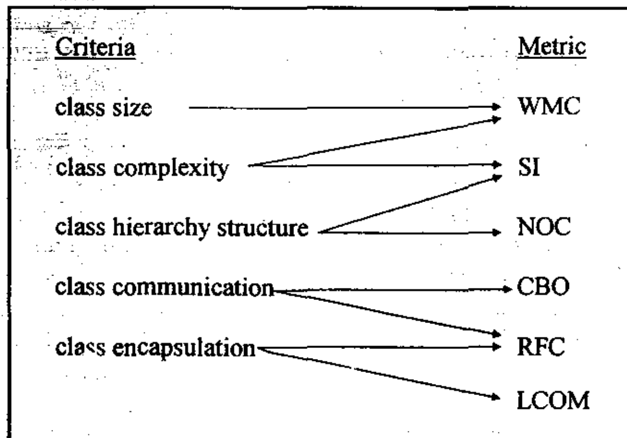


Figure 3. Dependencies between criteria and metrics.

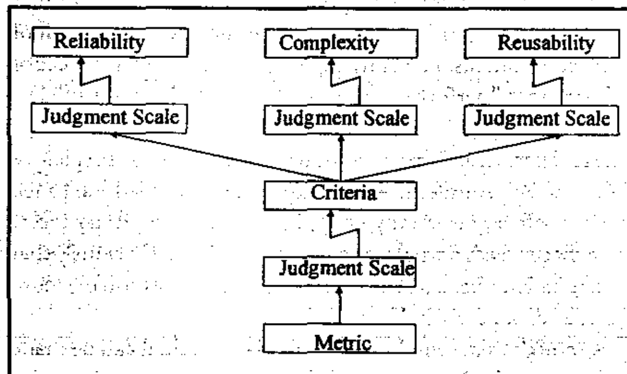


Figure 4. The process of computing scores for the three quality factors.

applies. Metrics may be a way forward. The metrics evolving for a particular design can guide the developer to refine the design to meet the quality factors outlined here. There are other software quality metrics such as readability, e.g., the number of commented methods, which this article hasn't addressed. ■

References

1. Card, D. N., and R. L. Glass. *Measuring Software Design Quality*, Prentice Hall, Englewood Cliffs, NJ, 1991.
2. Chen, J.-Y., and J.-F. Lu. "A New Metric for Object-Oriented Design," *Information and Software Technology*, 35(4): 232–239, 1992.
3. Chidamber, S. R., and C. F. Kemerer. "Towards a Metrics Suite for Object-Oriented Design," *ACM OOPSLA*, pp. 197–211, 1991.
4. Chidamber, S. R., and C. F. Kemerer. "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, 20(6): 476–493, 1994.
5. Fenton, N. E. *Software Metrics: A Rigorous Approach*, Chapman & Hall, London, 1992.
6. Graham, I. *Object-Oriented Methods*, Addison–Wesley, Wokingham, England, 1993.
7. Liu, K., Y. Ades, and R. K. Stamper. "Simplicity, Uniformity, and Quality: The Role of Semantic Analysis in Systems Development," *Proceedings of SQM '94 (Second International Conference on Software Quality Management)*, Edinburgh, Scotland, 1994.
8. Lorenz, M., and J. Kidd. *OO Software Metrics: A Practical Guide*, Prentice Hall, Englewood Cliffs, NJ, 1994.
9. Sommerville, I. *Software Engineering, fourth ed.*, Addison–Wesley, Wokingham, England, 1992.

ABOUT ABSTRACT CLASSES

continued from page 55

phase, we can say that the deferred class is a design notion. Our opinion about this subject is quite similar to Hürsch[5] but not exactly the same. We agree with the distinction between abstract and deferred classes established by Hürsch but, in our opinion, both kinds of classes, abstract and deferred, cannot be instantiated by themselves, and we can also find other kinds of noninstantiable classes that are neither abstract nor deferred.

In Figure 2 we propose a partial and overlapped generalization (partial in the sense that we can find noninstantiable classes that are neither abstract nor deferred, and overlapped in the sense that an abstract class can also be a deferred class, and vice versa). We have presented here the different cases of noninstantiable classes found in the literature: those (abstract classes) being introduced to increase the level of abstraction (and also to increase software reusability as a consequence); those introduced only to increase software reusability (deferred classes); those introduced because of model needs, such as the case of hierarchies of parallel inheritance; and finally, those introduced in some object models that require a single maximal supertype in each family of types. ■

Acknowledgments

We would like to thank James Rumbaugh for his courtesy in answering a question for us about the Unified Method. We would also like to thank Joaquín Cervera for his comments on an earlier version of this article.

References

1. Booch, G., and J. Rumbaugh. "Unified Method for Object-Oriented Development," *Documentation Set (V0.8)*, Rational Software Corporation, 1995.
2. Cattell, R. G. G., Ed., *The Object Database Standard: ODMG-93, Release 1.1*, Morgan Kaufmann, San Francisco, CA, 1994.
3. Henderson-Sellers, B. "Convergence Is in the Air," *Report on Object Analysis & Design*, pp. 47–49, Mar./Apr. 1996.
4. Henderson-Sellers, B. "The COMMA Project: First Steps," *Report on Object Analysis & Design*, pp. 49–52, May/Jun. 1996.
5. Hürsch, W. L. "Should Superclasses Be Abstract?" Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP '94), in *Lecture Notes of Computer Science (LNCS)*, M. Tokoro and R. Pareschi, Eds., Springer-Verlag, Berlin, 1994.
6. Chonoles, M. J., J. A. Schardt, and P. J. Magrogan. "Speaking of Methods," *Report on Object Analysis and Design*, pp. 8–12, Mar./Apr., 1996.
7. Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, NJ, 1988.
8. Melton, J. "Object Technology and SQL: Adding Objects to Relational Language," *Data Engineering IEEE*, 17(4): 15–26, Dec. 1994.
9. Spiby, P. *EXPRESS Language Reference Manual*, ISO TC184/SC4/WG5/P3, Apr. 1991.
10. Stroustrup, B. *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1991.
11. Wegner, P. "A Perspective on Object-Oriented Research," *Theory And Practice of Object Systems*, 1(2): 133–143.
12. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. *Design Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.
13. Soley, R.N., and C.M. Stone. *Object Management Architecture Guide*, Object Management Group (OMG), Framingham, MA, 1995.
14. Booch, G., J. Rumbaugh, and I. Jacobson. *Unified Modeling Language v1.0*, Rational Software Corporation, Santa Clara, CA, 1997.