Clusters of Classes: A Bigger Building Block

Donald G. Firesmith

For the past decade, Donald G. Firesmith has been providing consulting and training in object technology. He developed and maintains ADM, a fourth-generation object-oriented system and software development method; and OODSL, a second generation object-oriented specification and design language. A regular columnist for ROAD, he is also the author of OBJECT-ORIENTED REQUIREMENTS ANALYSIS AND LOGICAL DESIGN: A SOFTWARE ENGINEERING APPROACH (John Wiley and Sons, 1993), "The Dictionary of Object Technology" (SIGS Books, forthcoming), and "Object-Oriented Standards, Procedures, and Guidelines" (Prentice-Hall, forthcoming). He can be reached at Advanced Software Technology Specialists, 2910 Webster, Fort Wayne, IN 46807; 219.745.7928 (v); and 73664.3515@compuserve.com.

A TYPICAL APPLICATION INVOLVES dozens, hundreds, or even thousands of classes. As Grady Booch has often pointed out, classes by themselves are not large enough to deal effectively with the size and complexity of many applications. Some larger building block must therefore be used to decompose applications into meaningful, manageable pieces. This article discusses current terms and definitions that have been used to describe collections of classes and objects. Based on these definitions, this article recommends a standard definition of the term cluster and describes its properties. The article ends with a brief discussion of the role of clusters in an incremental, iterative development process.

CURRENT TERMS AND DEFINITIONS

Largely because languages have not typically provided such a building block, many methodologists have introduced the concept of a collection of classes and objects. Working in parallel, the methodologists have unfortunately also introduced numerous terms with either the same or subtly different meanings. In alphabetical order, these terms (and their proponents) have included the following:

- *Assembly* (Firesmith)
- *Class category* (Booch)
- *Cluster* (Eiffel, Firesmith, Lorenz, Meyer, Shlaer/Mellor)
- *Domain* (Shlaer/Mellor)
- *High-level object class* (Embley et al.)
- *Kit* (Berard)
- *Module* (Booch, Rumbaugh)
- *Pattern* (Coad)
- *Subject* (Coad/Yourdon)
- *Subsystem* (Booch; Jacobson; Lorenz; Shlaer/Mellor; Wirfs-Brock et al.)

The term *assembly* has been defined by Donald Firesmith[1] as "a logically-cohesive set of one or more *software* objects, classes, and subassemblies that are analyzed, designed, coded, tested, and integrated as a group, typically as the increment of development in a recursive development method. An assembly has an interface containing its protocol (visible features) and an implementation containing its body (hidden features). Those assemblies not contained within other assemblies are typically identified as system-level objects during system design, are major software configuration items, and typically implement a significant, cohesive set of system requirements"(p. 8). Notice that the term *assembly* was restricted to software; I use the term *subsystem* only when referring to a part of a system that typically contains hardware as well as software. The term *assembly* was chosen in the mid 1980s as the software equivalent of a hardware assembly. Because of the growing popularity of Eiffel and a desire for standardization, I have recently decided to use the term *cluster* with the original definition of assembly.

The term *class category* has been defined by Grady Booch[2] as "a logical collection of classes, some of which are visible to other class categories and others which are hidden. The classes in a class category collaborate to provide a set of services" (p. 512). Notice that Booch restricts the term *class category* to logical collections and uses the term *subsystem* for physical collections.

The term *cluster* has been defined by the Eiffel language and by Bertrand Meyer[3] as follows: "A *cluster* is a set of related classes" (p. 33). And, "Clusters correspond to the major divisions of a system" (p. 38). Clusters are not a syntactical part of Eiffel but are instead defined in the language for assembling classes in Eiffel (Lace). Lace, however, "is not part of Eiffel, and Eiffel environments are not required to support Lace" (Meyer,[3] p. 513). Firesmith[1] originally defined the term cluster as "a physical grouping of one or more modules and subclusters. A cluster has an interface containing its protocol (visible features) and an implementation containing its body (hidden features). Those clusters not contained within other clusters are typically identified as programs, are major software configuration items, and typically implement a significant, cohesive set of system requirements" (p. 13).* Mark Lorenz[4] defines the term *class cluster* as "a group of classes that collaborate a great deal with each other. Clustering them simplifies the

*Thus, while my term assembly corresponded to Booch's class category, my term *cluster* originally corresponded to Booch's *subsystem*.

amount of work required to test the functionality." Shlaer/Mellor[5] define the term *clusters* as "Groups of objects† that are interconnected with one another by many relationships." (p. 145). Notice that the term *cluster* has been defined as a collection of either classes (Eiffel, Firesmith, Meyer, Shlaer/Mellor), objects (Firesmith), modules (Firesmith), or all three (Firesmith).

The term *domain* has been defined by Sally Shlaer and Stephen Mellor[5] as follows: "A *domain* is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain" (p. 133). Note that the term *problem domain* should not be confused with *attribute domain, which* Shlaer and Mellor[6] define as follows: "The set of values an attribute can take on constitutes its *domain*" (p. 99).

The term *high-level object class* has been defined by David Embley, Barry Kurtz, and Scott Woodfield[7] as follows: "a *high-level object class* groups object classes, relationship sets, constraints, and notes into a single object class" (p. 99). According to Embley, high-level object classes can also group objects, although during analysis, objects only have identifiers, not attributes and operations.

The term *kit* has been defined by Ed Berard[8] as "a collection of objects (e.g., classes, metaclasses, nonclass instances, unencapsulated composite operations, other kits, and systems of interacting objects) all of which support a single large coherent object-oriented concept, e.g., windows, switches, and insurance policies" (p. 333).

The term *module* has been defined by James Rumbaugh et al.[9] as follows: "The lowest level sybsystems are called modules.... a coherent subset of a system containing a tightly bound group of classes and their relastionships."

The term *pattern* has been defined by Peter Coad[10] as follows: "A *pattern* is a template of interacting objects with stereotypical responsibilities, an example of an effective portion of an object model, one that may be applied again and again, by analogy."

The term *subject* has been defined by Peter Coad and Ed Yourdon[11] as follows: "A *subject* is a mechanism for guiding a reader (analyst, problem domain expert, manager, client) through a large, complex model"[11] (p. 106), and "Subjects are also helpful for organizing work packages on larger projects, based upon initial OOA investigations"[12] (p. 30). Notice that, unlike the other terms, subjects are not necessarily design components but typically vary from discussion to discussion. Subjects do not necessarily have anything to do with the increments of development, nor do they imply any encapsulation.

The term *subsystem* has many definitions. Grady Booch[2] defines the term *subsystem* as "a collection of modules, some of which are visible to other subsystems and others of which are hidden" (p. 519), whereby he defines the term *module* as "A unit of code that serves as a building block for the physical structure of a system; a program unit that contains declarations, expressed in the vocabulary of a particular programming language, that form the physical realization of some or all of the classes and objects in the logical design of the system" (p. 516). Note that unlike Rumbaugh, Booch does not explicitly define a module as a collection of classes.

Ivar Jacobson et al.[13] define the term *subsystem* as follows: "To be able to manage the system more abstractly, we use the concept of subsystem. A subsystem groups several objects. Subsystems may be used in both the analysis model and in the design model. Subsystems may also include other subsystems; the concept is recursive" (p. 143). A subsystem "groups objects and subsystems into managing units. The lowest level is called a service package and is an atomic managing unit. Subsystems may exist in the analysis model as well as in the design model" (p. 512).

Mark Lorenz[14] defines the term *subsystem* as "a group of classes that work together to provide a related group of functions. For example, an ATM application may have an interface subsystem that provides functions and

†Although their books imply that they use the term object to mean *class,* Mellor has stated that this is a mistake and that the term object means an *unspecified instance.*

Table 1a. Characteristics of clusters.

Methodologists	A Cluster is a Collection of: Objects	Classes	Clusters as parts	Attributes/ Operations	Non-OO Modules	Clusters Have Instances
Berard	yes	yes	yes	yes	yes	no
Booch	yes	yes	yes	no	only if necessary	yes
Coad	yes	yes	yes	no	no	yes
Embley et al.	yes	yes	yes	yes	no	yes
Firesmith	yes	yes	yes	no	only if necessary	yes
Jacobson et al.	yes	nc	yes	no	no	no
Lorenz	nc	yes	yes	yes	only via wrappers	no
Meyer (Eiffel)	nc	yes	yes	no	only if necessary	no
Rumbaugh et al.	yes	yes	yes	no	no	no
Shlaer and Mellor	yes	yes	no	no	no	no
Wirfs-Brock et al.	nc	yes	yes	no	no	no

Table 1b. Characteristics of clusters.

Methodologists	Cluster Inheritance	Cohesive Abstraction	Component of Design	Supports Encapsulation	Increment of Development
Berard	no	yes	yes	no	maybe
Booch	yes	yes	yes	yes	maybe
Coad	no	yes	yes	no	yes
Embley et al.	no	yes	no	no	maybe
Firesmith	yes	yes	yes	yes	typically
Jacobson et al.	no	yes	yes	no	yes
Lorenz	no	yes	yes	yes	typically
Meyer (Eiffel)	yes	yes	yes	yes	yes
Rumbaugh et al.	no	yes	yes	no	maybe
Shlaer and Mellor	no	yes	yes	maybe	no
Wirfs-Brock et al.	no	yes	yes	yes	maybe

Clusters should be used as an increment of development in an incremental, iterative, parallel object-oriented development cycle. The classes in a cluster should be analyzed, designed, coded, tested, integrated, and documented as a group.

classes related to communication to and receiving inputs from the end user" (p. 219).

Sally Shlaer and Stephen Mellor[5] define the term *subsystem* as follows: "When partitioning a domain, we divide the information model so that the clusters remain intact.... Each section of the information model then becomes a separate subsystem" (p. 145).

Rebecca Wirfs-Brock et al.[15] define the term *subsystem* as follows: "A subsystem is a set of classes (and possibly other subsystems) collaborating to fulfill a set of responsibilities, as part of a larger piece of software.... Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts" (pp. 36, 135). Subsystems, like clusters, can contain either classes,[4,14,15] objects,[13] or modules.[2]

Although the term *subsystem* is slightly more popular among methodologists than the term *cluster*, it unfortunately has significant potential problems. The term subsystem has traditionally been used to refer to a part of a *system*, which typically contains hardware, software, wetware (i.e., people), and paperware (i.e., documentation). When O-O methodologists use the term, however, they typically intend only a collection of *software* objects or classes. Thus, the same term is used for two quite different things at two different levels of abstraction that are typically developed by two different groups. Software decomposition need not result in the same collections as system decomposition. The use of the same term on projects developing systems containing software can lead to confusion and misunderstandings.

RECOMMENDED CLUSTER DEFINITION

Based on a review of current terms for collections of objects and/or classes, I would like to recommend that the object community standardize on the term *cluster*. For the sake of the software engineering principle of uniformity, I also recommend that the same term be used for both logical and physical collections as well as for both pure object-oriented collections and hybrid collections containing both object-oriented and non-object-oriented units.

Regardless of the terms used, Table 1 summarizes the key concepts from the books listed in the References section and from phone calls with the methodologists involved. Based on a review of the definitions of these current terms and of reasonable extensions that will make the concept of a cluster more powerful, a cluster can be viewed as analogous to an aggregate class used to encapsulate other classes and clusters but *without* any attributes and operations. The key concepts that should either be included in the definition of the term *cluster* or used to describe good clusters are the following:

- Clusters may be used to localize the following kinds of cluster features:
  - Classes.
  - Clusters as component parts (a.k.a. subclusters‡).
  - Invariants that involve multiple encapsulated classes or component clusters.
  - Non-object-oriented modules if necessary when dealing with hybrid languages or legacy software. However, clusters should typically *not* be used to localize *class* features such as the following:
    - *Properties*, because this would reintroduce common global data with their associated problems.
    - *Behavior*, because this would reintroduce common global operations with their associated problems.
    - *Messages* to clusters, because this would require cluster-level operations (see above).
- Like classes, clusters should be able to be parameterized by class, component cluster, invariant, or module (a.k.a. generic clusters).
- Each cluster can be instantiated to produce a collection of *objects*, instances of other clusters as components, and non-object-oriented modules (if necessary due to the use of a hybrid language or legacy software).
- Clusters should support information hiding and scoping rules. Instances of clusters may export some of their features and hide others. For example, objects in one cluster instance should be able to send messages to other objects in the same cluster instance and to visible objects in the protocols of other cluster instances, but not to objects hidden in the implementations of other cluster instances.
- Clusters should be logically cohesive, capturing a single higher-level abstraction, scenario, or cohesive set of requirements, responsibilities, or capabilities.
- Clusters should be loosely coupled to other clusters, with the internal coupling between cluster features greater than their coupling to the visible features of other clusters.
- Clusters should be the unit of cluster inheritance; this allows new derived clusters to inherit the features of one or more existing base clusters.
- Clusters should be used as an increment of development in an incremental, iterative, parallel object-oriented development cycle. The classes in a cluster should typically be analyzed, designed, coded, tested, integrated, and documented as a group by a small team of developers.
- The scope of an object-oriented diagram documenting more than a single class is often a single cluster instance or cluster, possibly including terminators. For example, a semantic net or extended entity relationship diagram may document a cluster of classes and their associations. Similarly, an interaction diagram may document the message paths between objects in a representative instance of a cluster.
- The concepts of cluster instance, cluster, and cluster inheritance should be supported and standardized by object-oriented programming languages so that the same language support provided to objects and classes is also available to clusters. Clusters should not be relegated to an optional library feature that varies from vendor to vendor.

Based on the ideas and terms used with objects and classes in the preceding discussion,

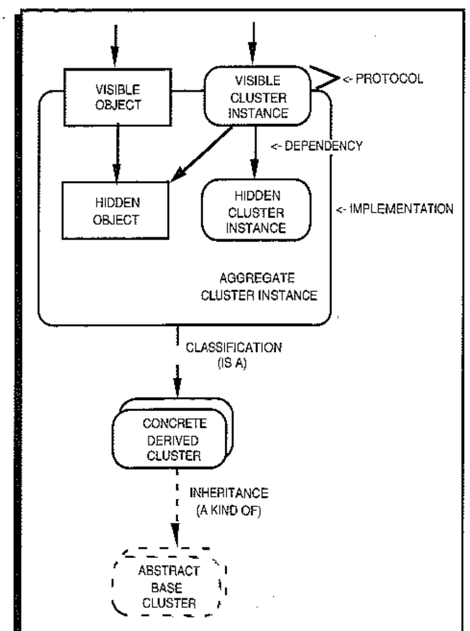

Figure 1. Cluster concepts.

‡The term *subcluster* is questionable because of the desire to use inheritance with clusters.

I currently endorse the following definitions of the terms relating to clusters:

An *instance of a cluster* (a.k.a. *cluster instance)* is a uniquely identified cohesive collection of one or more *objects*, component cluster instances, and non-object-oriented modules (if necessary due to the use of a hybrid language or legacy software). A cluster instance is an instance of one or more clusters. A cluster instance is an encapsulation with an interface containing its protocol of visible features (e.g., objects) and an implementation containing its body of hidden features. A cluster is similar to an aggregate object but without its attributes and operations. (See Fig. 1.)

A *cluster* is a uniquely-identified, cohesive collection of one or more *classes*, component clusters, and non-object-oriented modules (if necessary due to the use of a hybrid language or legacy software). A cluster is typically developed and documented as a group. A cluster is an encapsulation with an interface containing its protocol of visible (i.e., public) features and an implementation containing its body of hidden (i.e., protected or private) features. A cluster is the unit of cluster inheritance (i.e., a specification of features that may be reused by other clusters via inheritance). A cluster is an implementation of a cluster type. A cluster is similar to an aggregate class but without attributes and operations.

A *generic cluster* is a cluster that has been parameterized (e.g., with classes, component clusters, invariants).

A *cluster type* is a uniquely-identified abstraction (i.e., model) of one or more *types*, component cluster types, and non-object-oriented modules (if necessary due to the use of a hybrid language or legacy software). A cluster type specifies the protocols of its exported features (i.e., it is a type whose instances are clusters rather than objects.) A cluster type is the unit of cluster subtyping (i.e., a specification of visible features that may be reused by other cluster subtypes).

Cluster inheritance is the incremental definition of new derived clusters that inherit the features of one or more existing base clusters (i.e., inheritance between clusters). A *derived cluster* is a cluster that inherits from one or more other clusters, and a *base cluster* is a cluster from which other clusters inherit. A derived cluster can either add new features (e.g., classes or subclusters) or override inherited features (e.g., replace an inherited class with a descendent class). (See Fig. 2.)

An *aggregate cluster* is a cluster that contains another cluster as a component part, whereas an *atomic cluster* does not contain any clusters. In the tradition of subsystems and substates, a *subcluster* is a component cluster within another cluster.

CLUSTER DEVELOPMENT

Because clusters, unlike classes, do not necessarily correspond to any one kind of thing in the application domain, different developers may decompose a single application into clusters in different ways. System-level clusters may therefore not correspond exactly to software-level clusters, although it is certainly convenient when they do. If clusters are incrementally developed as part of an incremental iterative object-oriented development cycle, their structure may be a historical accident.

For example, a new cluster may be developed when it is discovered that classes in the cluster currently being developed depend on other classes that have not yet been identified.

Using such an approach, there is no guarantee that the resulting cluster(s) will be highly cohesive or minimize cross-cluster coupling. If clusters are identified based on scenarios (e.g., use cases), mechanisms, or functional requirements, then they will probably exhibit functional rather than object cohesion. This can cause different teams developing different clusters to unintentionally develop different (and partially redundant) variants of the same class. For all these reasons, clusters may have a highly iterative life cycle.

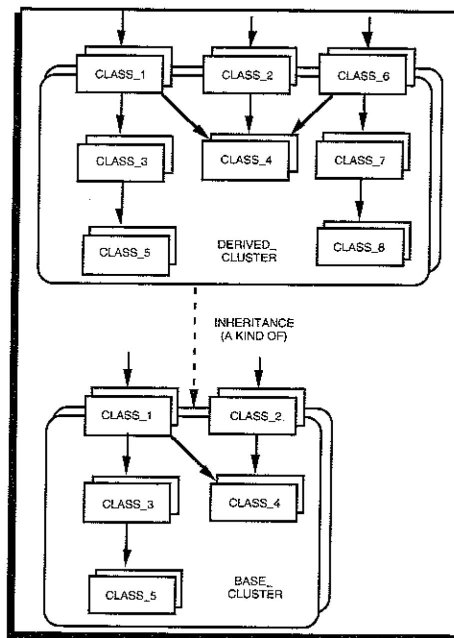

Figure 2. Cluster inheritance.

An architect should make an initial decomposition of an application into its primary clusters. These clusters may then be incrementally developed, with new clusters identified as needed to support existing clusters. Clusters may be incrementally developed either top-down or outside-in by message passing or dependency. To produce a final architecture, the cluster boundaries may be incrementally optimized to minimize coupling, maximize cohesion, and optimize cluster inheritance.

CONCLUSIONS

Because applications often consist of numerous objects and classes, some building block larger than a class must therefore be used to decompose applications into meaningful, manageable pieces. That building block is the cluster, an encapsulation of logically and physically related classes and subclusters that are often analyzed, designed, coded, and tested together as a small increment of development. Although one cannot send messages to clusters because they do not (and should not) have attributes and operations, clusters may enforce associated business rules (i.e., invariants). One should be able to instantiate clusters to produce groups of related objects, and cluster inheritance should allow developers to drive new clusters from existing base clusters. Finally, clusters, cluster instances, and cluster inheritance should be supported by object-oriented programming languages and upperCASE tools as well as by analysis and design methods.

References

1. Firesmith, D. ASTS Abbreviations and Glossary, ASTS Goi-Version 1.8, Advanced Software Technology Specialists, Fort Wayne, IN, 4 June 1994.
2. Booch, G. Object-Oriented Analysis and Design with Application (2nd ed.), Benjamin/Cummings, Menlo Park, CA, 1994.
3. Meyer, B. Eiffel: The Language, Prentice Hall, Englewood Cliffs, NJ, 1992.
4. Lorenz, M. Rapid Software Development with Smalltalk, SIGS Books, New York, NY, 1995.
5. Shlaer, S. and S. Mellor. Object Lifecycles: Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1992.
6. Shlaer, S. and S. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1988.

continued on page 25

ier to systematically derive an implementation from a model-based specification or show that an implementation conforms to a model-based specification. From the model perspective, algebraic methods are suitable for specifying the object model, model-based specifications are suitable for the functional model, and state-transition methods[9,16] are suitable for the dynamic model. Such an approach requires precise guidelines for systematically ensuring consistency among the models and for using the formal specifications to show that the requirements are satisfied.

References

1. Booch, G. Object-Oriented Analysis and Design with Applications, Benjamin Cummings, Redwood City, CA, 1994.
2. Coad, P. and E. Yourdon. Object-Oriented Analysis, Yourdon Press, New York, 1991.
3. D. Coleman et al. Object-Oriented Development: The Fusion Method, Prentice Hall, 1994.
4. I. Jacobson et al. Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison-Wesley, Reading, MA, 1992.
5. Rumbaugh, J. et al. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
6. Shlaer, S. and S. Mellor. Object-Oriented system analysis, Yourdon Press, Englewood Cliffs, N.J., 1988.
7. Alencar, A. and J. Goguen. OOZE: An object-oriented Z environment, ECOOP'91, pp. 180–199, 1991.
8. Carrington, D. Object-Z: An object-oriented extension to Z. Formal Description Techniques, December 1989.
9. Coleman, D. et al. Introducing objectcharts, or How to use statecharts in object-oriented design, IEEE Transactions on Software Engineering 18(1):9–18, 1992.
10. Feijs, L. and H. Jonkers. Formal Specification and Design, Cambridge University Press, New York, 1992.
11. Van Leeuwen, J. The RAISE Specification Language, MIT Press, Cambridge, MA, 1990.
12. Wieringa, R. Steps towards a method for the formal modeling of dynamic objects, Data and Knowledge Engineering, vol. 6, 509–540, 1991.
13. Cook, S. and J. Daniels. Designing Object Systems, Prentice Hall, Englewood Cliffs, NJ, 1994.
14. Wordsworth, J. Formal Specification in Z, Addison-Wesley, Reading, MA, 1992.
15. Dodani, M. Specifying object-oriented software, ROAD (1)3:33–36, 1994.
16. Harel, D. Statecharts: A visual formalism for complex systems. Science of Computer Programming 8:231–274, 1987.

Mahesh Dodani is with the Department of Computer Science, The University of Iowa, Iowa City, IA 52242; (319) 335-0713 (v); (319) 335-3624 (f); dodani@cs.uiowa.edu (email).

VIEWS ON MODELING, *continued from page 21*

7. Embley, D. B. Kurtz, and S. Woodfield. Object-Oriented Systems Analysis: A Model-Driven Approach, Prentice Hall, Englewood Cliffs, NJ, 1992.
8. Berard, E. Essays on Object-Oriented Software Engineering, (vol. 1), Prentice Hall, Englewood Cliffs, NJ, 1993.
9. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
10. Coad, P., D. North, and M. Mayfield. Object Models: Strategies, Patterns, and Applications, Prentice Hall, Englewood Cliffs, NJ, 1995.
11. Coad, P. and E. Yourdon. Object-Oriented Analysis (2nd ed.), Prentice Hall, Englewood Cliffs, NJ, 1990.
12. Coad, P. and E. Yourdon. Object-Oriented Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
13. Jacobson, I., M. Christerson, P. Jonsson, and G. Oevergaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokingham, England, 1992.
14. Lorenz, Mark. Object-Oriented Software Development: A Practical Guide, Prentice Hall, Englewood Cliffs, NJ, 1993.
15. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, NJ, 1990.

CLIENT SERVER DEVELOPER

Call for Papers!

CLIENT/SERVER DEVELOPER is a new publication committed to helping programmers, developers and technical managers understand C/S technology. We are now actively seeking manuscripts on the following:

Operating Systems • Databases • Programming Languages • Object Technology and Reuse • C/S Application Design Methodologies and Tools • Software Engineering Methodologies • Pre-Packaged C/S Applications • Business Process Re-Engineering • Project Management in a C/S Environment • Metrics and Testing • Multimedia

To submit an article or request author guidelines, contact:

Thomas O'Flaherty, Editor
411 West End Avenue, Suite 2B
New York, NY 10024
Phone: 201.801.0050 Fax: 201.801.0441

Published by SIGS PUBLICATIONS