# The Many Techniques Used to Identify Objects and Classes

## or, All roads may lead to Rome, but some are shorter and safer

Donald G. Firesmith

Donald G. Firesmith provides consulting and training in object technology and is the developer of ADM and OOSDL. He is the author of Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach (John Wiley and Sons, 1993). He is currently writing two books: Dictionary of Object Technology and Testing Object-Oriented Software (to be published by SIGS Books in 1995). He can be reached at Advanced Software Technology Specialists, 2910 Webster, Fort Wayne, IN 46807, 219.745.7928, 73664.3515@compuserve.com.

IDENTIFYING OBJECTS AND CLASSES is a central step of all major object-oriented development methods. And identifying them should be easy. After all, we deal with hundreds of real-world objects and classes every day. Yet it is often difficult, especially when one is trying to identify good objects and classes that will properly collaborate together to implement a specific mechanism or application.

This article—an updated summary of Chapter 4 from my first book[1]—discusses the many techniques that have been developed over the years to identify objects and classes. It also categorizes these techniques and discusses their individual strengths and weaknesses. In addition, the identification process and the sources of information needed for identification are covered.

### THE IDENTIFICATION PROCESS

Most object-oriented development methods contain a step in which the objects and classes are to be identified. However, most applications contain hundreds of objects and classes, far too many for any individual engineer or team to identify at once. Fortunately, most methods also incorporate a development cycle that is iterative, incremental, and parallel—a cycle in which the application is analyzed, designed, programmed, tested, and integrated in small increments (a.k.a. assemblies, clusters, subsystems) by small teams working in parallel. What is described as a single identification step in the books and articles is actually an activity that is typically performed many times by several teams over a large portion of the project schedule. Objects and classes may also be identified and developed as part of larger increments such as horizontal layers, vertical partitions, builds, and releases. Conversely, inheritance structures are typically extended, one class at a time.

The direction of incremental development (and identification) varies from method to method and from engineer to engineer. In terms of message passing, incremental development can be performed either top-down (i.e., clients before servers), bottom-up (i.e., servers before clients), or outside-in starting with objects modeling terminators. In terms of reuse and the inheritance structures, the identification process is primarily performed from the bottom up, extending general superclasses with specialized subclasses.*

A common technique is to divide the application into horizontal layers, as in Figure 1. For example, the domain layer contains model objects and classes that model application-domain things, whereas the GUI layer contains view, presentation, and other domain-independent user interface objects and classes. When adequate domain expertise is available, models are often identified before their associated views and presentations, because these latter objects are defined in terms of (and must send messages to) their models. Conversely, presentations may be identified and analyzed before their models, because users can often specify the GUI screens that they want to use easier than they can specify the underlying models that the screens access.

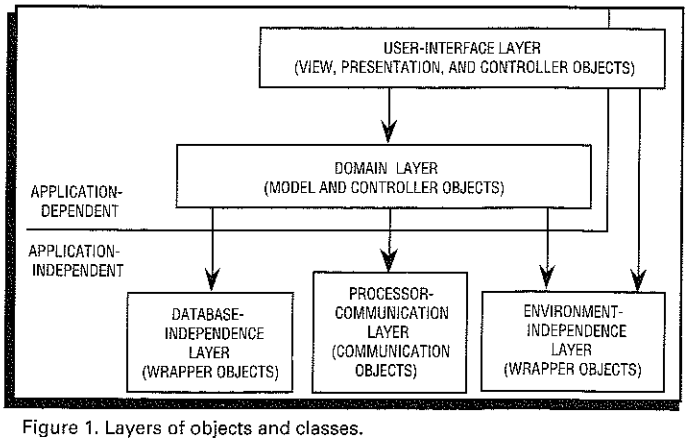


Figure 1. Layers of objects and classes.

### SOURCES OF INFORMATION

As mentioned in my first ROAD column,[2] there are many sources of information that can be used to identify objects and classes. These include:

- People such as
  - Application domain experts
  - Users

* Because of iteration, the actual process is rarely as linear as described here. For example, domain experts often recognize concrete classes before abstract classes, and one must often optimize the inheritance structures by adding new intermediate classes.

- Customers
- Documentation for the current application or any similar existing application, such as
  - Requests for Proposals (RFPs)
  - System Requirements Specifications
  - System Design Documents
  - Software Requirements Specifications
  - User's manuals
  - Training manuals and materials
  - Data dictionaries

  Unfortunately, the information concerning any one object or class may be scattered throughout the source document if functional decomposition was used to create it.
- Engineer's own experience
- Reuse repositories and class libraries

## IDENTIFICATION TECHNIQUES

Although a large number of identification techniques have been developed during the last 15 years, only a small number are typically taught as part of most object-oriented development methods. Some identification techniques are relatively obsolete (though still advocated in books and articles), whereas others are relatively state-of-the-art. Some techniques require significant expertise to use effectively, whereas others are quite indirect and are best used by those who have not yet made the transition to the object-oriented mindset. A few techniques are tightly coupled with specific object-oriented development methods, whereas most techniques are quite general and can be used with any major development method. Some techniques are highly reliable, whereas others are relatively unreliable, have many associated problems, and produce numerous false identifications.

This article divides the various identification techniques into three main categories:

P) Primary techniques
S) Secondary techniques
T) Traditional techniques

## PRIMARY TECHNIQUES

The primary techniques are generally the most powerful techniques to use. They identify the most objects and classes, are relatively direct, and do not produce many false identifications. Unfortunately, these techniques usually require the engineer to have at least a minimal acquisition of the object mindset. Developers should therefore concentrate on mastering these techniques and use the secondary techniques primarily for verification purposes.

The primary techniques can be divided into the following two subcategories:

- Those designed for *initial identification* during an organization's initial projects
- Those designed for *reidentification* on subsequent projects

Primary identification techniques designed for the initial identification of objects and classes include using:

PI-1) The definitions of the terms *object* and *class*
PI-2) The different categories of domain objects and classes
PI-3) Terminators on context diagrams
PI-4) Nodes on whiteboards
PI-5) Generalization and specialization
PI-6) Inheritance and classification
PI-7) Scenarios
PI-8) Object decomposition
PI-9) Specification and design languages

*PI-1) Definitions.* An *object* is defined as a model of a thing in the application. A *class* is defined as a template for the instantiation of similar objects and, therefore, a class models a set of related things rather than an individual thing. After using object technology for awhile, engineers develop a feel for what objects and classes are. They are then able to identify objects and classes by intuitively recognizing them when they see them.

*PI-2) Categories of application-domain-specific objects and classes* have been published by many methodologists (see Fig. 2). Engineers can use lists of these categories to identify objects and classes by asking themselves, "Do I need any of these in my application?" Following and extending the Smalltalk model-view-controller idiom, the ASTS Development Method (ADM)[1] divides domain objects and classes into the following categories and subcategories:

- **Model objects** that model application domain things. Sometimes referred to as *business* or *core objects*, these key abstractions can be further subdivided into the following:
  - *Concepts*, such as altitudes, distances, pressures, speeds, temperatures, times, volumes, and weights
  - *Devices*, such as actuators, buttons, motors, sensors, switches, and valves
  - *Documents*, such as articles, books, certificates, licenses, and reports
  - *Events*, such as accidents, failures, and launches†

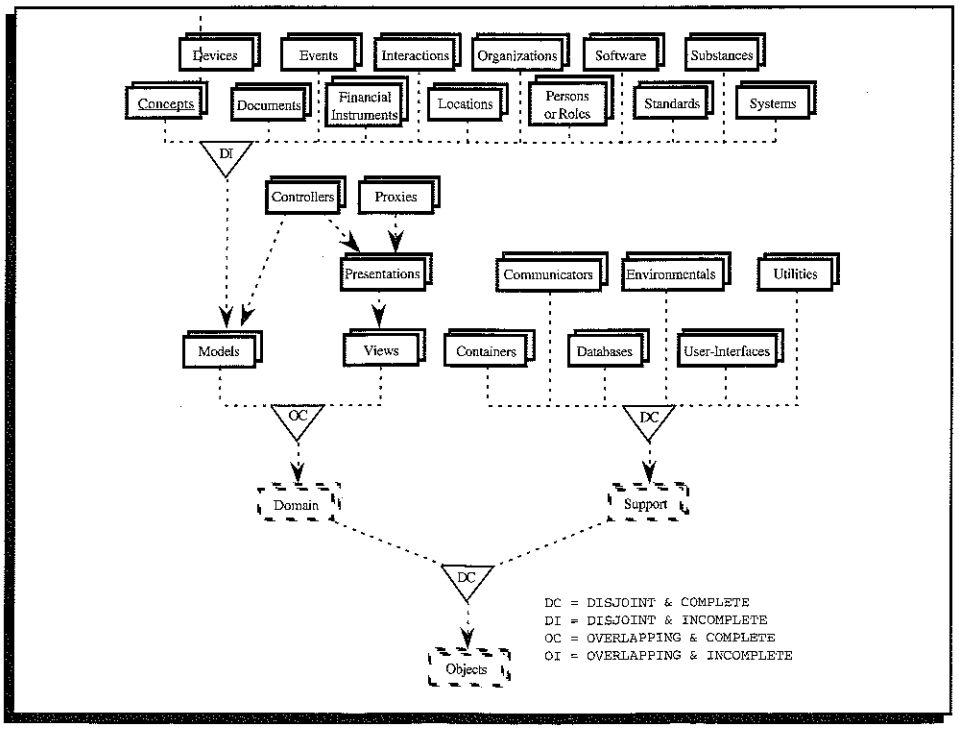


Figure 2. Categories of objects and classes to be identified.

† When identifying event, interaction, and standards objects, engineers should be careful to ensure that functions are not improperly masquerading as objects. These subcategories of objects should model things rather than operations. If they truly need to be modeled as objects, then they would best be described by nouns rather than verbs, and they should have their own attributes, operations, and—possibly—states. Care should also be taken when the abstraction of the object could be described with both a noun and a verb. For example, a customer may *purchase* an item (verb implies operation) or a *purchase* of an item may be logged into a purchase-order system (noun implies object).

  - *Financial instruments,* such as bonds, commodities, money, and stocks
  - *Interactions,* such as contracts, divorces, loans, marriages, and purchases
  - *Locations,* such as airports, cities, drilling sites, offices, street intersections, and waypoints
  - *Organizations,* such as businesses, companies, departments, divisions, military units, and teams
  - *Persons and roles played,* such as customers, employees, operators, persons, tellers, and users
  - *Software* that is interfaced with, e.g., applications, databases, legacy systems, and operating systems
  - *Standards,* such as business processes, control algorithms, document templates, and recipes
  - *Substances,* such as air, chemicals, oil, and water
  - *Systems* containing hardware, software, wetware (i.e., people), and paperware (i.e., documentation), such as computers and networks
- **View objects,** i.e., those providing a view of the contents of one or more model objects
- **Presentation objects**—view objects that format and present a view to the user. Presentation objects are typically GUI view objects used to interface with users of the application
- **Proxy objects**—view objects that represent model objects residing on another processor
- **Controller objects** that exist to control one or more objects. Controller objects are often either model objects that model input devices (e.g., mice) or are presentation objects allowing the user to control the model objects

Based on the Door Master[§] requirements presented in my previous column,[3] many of the necessary objects model the following:

- **Concepts**
  - The _Entry_ code
  - The _Security_ Code
- **Devices**
  - The _Change_Entry_Code_ Button
  - The _Change_Security_Code_ Button
  - The _Disable_ Button
  - The _Disabled_ Light
  - The _Door
  - The _Door_Lock
  - The _Door_Sensor
  - The _Enable_ Button
  - The _Enabled_ Light
  - The _Numeric_ Keypad
  - The _Speaker

*PI-3) Terminators on context diagrams* are often excellent choices for objects and classes. Model objects can be identified to handle the interface with these terminators, localizing the impact of changes to a terminator to the individual object that interfaces with it. Engineers should remember to model important indirect terminators as well as immediate terminators. For example, the software may indirectly interface with a sensor via an analog-to-digital converter, and both terminators should be modeled as objects.

*PI-4) Node cards on whiteboards* can be used to trigger the engineer's recognition of missing objects and classes in object-oriented diagrams under development. This is a common technique used during JAD/RAD requirements elicitation and modeling sessions.[2] Given a partially completed object-oriented diagram, the engineer answers the question, "What *nodes* are missing on this diagram?"

*PI-5) Generalization and specialization* can and should be used to identify objects and classes. Generalization can be used to identify classes from existing instances, superclasses from existing subclasses, and abstract and generic superclasses from concrete subclasses. Conversely, specialization can be used to identify instances from existing classes and subclasses from concrete, abstract, or generic superclasses. Given a class, the engineer can answer the question, "What are its more general superclasses and more specific specializations and instances?"

*PI-6) Inheritance and classification* can and should be used like generalization and specialization to identify objects and classes.[‖] Inheritance can be used to identify (concrete) subclasses from existing (abstract) superclasses and (abstract) superclasses from existing (concrete) subclasses. Classification can be used to identify classes from existing instances and instances from existing classes. Given an existing class, the engineer can answer the question: "What are its subclasses, superclasses, and instances?"

*PI-7) Scenarios* (e.g., use cases) are a popular source of objects and classes to be identified,[3] especially since the publication of Jacobson's Object-Oriented Software Engineering method.[4] Although a good technique for eliciting *functional* requirements, capturing mechanisms, and preparing integration tests, scenarios have several associated risks when it comes to using them to identify objects and classes.

Scenarios are functional abstractions. The mapping between scenarios and objects and classes is definitely not one-to-one. A single scenario typically involves only some of the resources of multiple objects and classes, and a single object or class may be involved in the implementation of multiple scenarios.

Large projects typically involve many scenarios, which are often divided among several teams for analysis and design. Different teams may redundantly identify multiple partial variants of the same object and class, and it is difficult to minimize this during the rush of real projects. For this reason, I recommend using scenarios primarily as a verification technique to ensure that all objects and classes have been identified rather than as an initial identification technique.

The Door Master scenarios presented in my second column[3] mentioned and could also have been used to identify the objects that I previously listed following identification technique PI-2.

*PI-8) Object decomposition* is a useful identification technique when dealing with aggregate objects. Although functional decomposition was the primary identification technique used by structured methods to identify new functions, object decomposition should not be used as the primary identification technique by object-oriented methods. Only the European Space Agency's Hierarchical Object-Oriented Development (HOOD) method is based on this approach.[#] For each object, the engineer should answer the question, "Is it a part of some larger aggregate?" and for each aggregate object or class, the engineer should answer the question, "What are its *relevant* component parts?"

[§] Door Master is a system that controls access of employees through a secured door such as might be found in an airport or company facility.

[‖] PI-5 and PI-6 are highly related but take different viewpoints. PI-5 is primarily used during analysis, whereas PI-6 is more oriented toward design.

[#] HOOD, which was developed for Ada83, is not really an object-oriented method as it does not support inheritance. It also uses object decomposition because it does not have the concept of assembly, cluster, or subsystem that is needed for incremental development.

*PI-9) Specification and design languages* provide a more formal way of incrementally identifying objects and classes than the somewhat obsolete lexical analysis approach that will be discussed under the traditional techniques. Instead of writing a narrative English paragraph and underlining the nouns, engineers can instead identify new objects and classes as they specify and design the corresponding code using an object-oriented specification and design language (e.g., OOSDL[1]) or use an appropriate implementation language. Because of its readability and inclusion of assertions, Eiffel makes an excellent specification and design language. While not as powerful as Eiffel, Smalltalk is also a reasonable choice for projects that will be implemented in Smalltalk. Because of its lack of assertions and readability as well as its relatively low-level of abstraction, C++ is definitely *not* appropriate as a specification language.

Organizations that have already developed their first object-oriented application should not have to identify their objects and classes from scratch on subsequent projects. Many of the relevant objects and classes should already have been identified and developed on previous projects. Engineers must then either reidentify and reuse them or use larger building blocks (e.g., frameworks) that contain and hide them, so that these encapsulated objects and classes do not need to be identified. Primary identification techniques designed for the *reidentification* of objects and classes include using the following:

PR-1) Object-oriented domain analysis and enterprise modeling
PR-2) Repositories of previously developed code
PR-3) Nodes on O-O diagrams
PR-4) Personal experience

*PR-1) Object-Oriented Domain Analysis (OODA) and enterprise modeling* may have already identified the majority of the objects and classes needed on the current project. Identification consists primarily of learning and browsing the results of the OODA, looking for relevant objects and classes.

*PR-2) Repositories of previously developed code* is also a source of objects and classes for use on the current project. These may consist of commercial class libraries, class libraries developed in-house, or frameworks of collaborating objects and classes. Once again, learning and browsing the existing software allows one to easily reidentify objects and classes for the current application. With application frameworks that can be treated as large software black boxes, one can even avoid the identification of internal objects and classes merely by using the framework. Thus, the most efficient way to identify objects and classes is to make their identification unnecessary.

Unfortunately, most organizations do not yet have large repositories of existing objects and classes, and those that do often do not have adequate browsing tools for finding the necessary objects and classes among the many unnecessary ones.

*PR-3) Nodes on existing relevant object-oriented diagrams* may well be objects and classes that belong in the current application. Unfortunately, such diagrams do not often exist prior to the need to identify objects and classes.

*PR-4) Personal experience* may also be a great source of objects and classes. Engineers who have previously worked in the same application domain will remember (and therefore reidentify) objects and classes that they used before. Because the personal experience of domain experts is also a prime source of objects and classes, joint application development (JAD) sessions with domain experts are very effective.[2]

## SECONDARY TECHNIQUES

The secondary techniques are typically indirect; if something usually corresponds to an object or class and if you can find that thing, then you have also indirectly found the corresponding object or class. These techniques are best used either (1) by those who are new to object technology and who do not yet have the experience to effectively use the primary techniques, or (2) as backup techniques to verify the results of the primary techniques.

Secondary identification techniques for the identification of objects and classes include using the following:

S-1) Abstract data types (ADTs)
S-2) Abstract state machines (ASMs)
S-3) States
S-4) Resources (a.k.a. features)
S-5) Parameters of messages
S-6) Associative objects
S-7) Role objects
S-8) Requirements
S-9) Class-responsibility-collaboration (CRC) cards
S-10) Entities on entity relationship attribute diagrams

*S-1) Abstract data types (ADTs)* can be used to indirectly identify classes. With partial justification, some authors have written that object-oriented programming is little more than programming with ADTs. Engineers who know when to use ADTs can therefore be reasonably confident that they have also indirectly identified a class corresponding to the ADT. However, ADTs are based more on data abstraction than on object abstraction and fail to take into account that objects consist of more than data and associated operations. If an object contains attributes of more than one simple ADT, then identifying a class for each ADT may in fact be misidentifying only a part of a class as a class.

*S-2) Abstract state machines (ASMs)* can be used to indirectly identify objects. In the object paradigm, all states are states of objects (or possibly classes or extents of classes). Therefore, each ASM can typically be mapped to its corresponding object (or class or extent).

*S-3) States,* like state machines, can be used to indirectly identify the object, class, or extent that they describe. For every state, the engineer should answer the question, "What is this the state of?"

*S-4) Resources* (a.k.a., features, members) can be used to indirectly identify objects and their classes. There is no such thing in the object paradigm as common global data or common functions not encapsulated within objects and classes. Similarly, every message must be sent from one object to another object or class. Exceptions must be identified, raised, and handled by objects. Invariants must involve the attributes of one (or more objects). The engineer should answer the following questions:

- "For all data, what object must calculate or store the data as an *attribute?*"
- "For each *operation,* what object is responsible for executing it and what objects are responsible for requesting that it be executed?" A danger with using operations is that they may be aggregates that are not allocatable to individual objects.
- "For all communication, what objects are responsible for sending the *messages* and what object or class is responsible for receiving them?"
- "For all error conditions, what object is responsible for identifying it (i.e., raising

the *exception*) and what objects must be notified (i.e., handle the exception)?"
- "For every (business) rule, what object(s) are responsible for enforcing it (e.g., as an *invariant*)?"

*S-5) Parameters of Messages* can be used to indirectly identify objects and their classes. For every message, the engineer should answer the question, "What parameters (i.e., objects) must flow with the message, what are their classes, and what is the class of the returned value (if any)?"

*S-6) Associative Objects* (and their associated classes) can be identified for many-to-many associations, for associations with attributes and operations, and for ternary (and higher order) associations.

*S-7) Role Objects* (and their associated classes) can be identified when the same object must play multiple roles in an application.

*S-8) Requirements* can be used to indirectly identify objects and classes that implement them. You may be surprised to see this method so low in the listing, but unfortunately, using requirements directly has may drawbacks.

Requirements specifications are almost always incomplete, inconsistent, vague, and not organized along object lines. Like scenarios, requirements are often functional abstractions. Several objects must usually collaborate to implement any one requirement, and the same object may be involved in the implementation of multiple requirements. Thus, objects and classes do not typically map one-to-one to requirements, and objects and classes identified by requirements may be incomplete, with multiple partial variants of each class being developed by different teams from different subsets of the requirements. Beware of functions masquerading as objects.

*S-9) Class-Responsibility-Collaboration (CRC)* cards are often cited as a way of identifying classes, although their primary purpose is informally documenting already identified classes. As with node cards (covered by technique PI-4), CRC cards will often force the identification of new classes, especially as collaborators of existing classes. As mentioned in my first ROAD article,[2] I have found node cards to be a more effective technique because of their use in conjunction with whiteboards in depicting graphical relationships.

*S-10) Entities on Entity Relationship (ER) Diagrams* quite often correspond to classes in an object-oriented design. If such ER diagrams already exist, then they may be used to identify classes as long as engineers remember that data modeling and object modeling are not exactly equivalent, especially if the data model has been normalized for implementation using a relational database. The extent of classes of complex objects must often be stored in multiple normalized tables,** and if entities are equated with tables, then they may not map one-to-one to classes.

## TRADITIONAL TECHNIQUES

The traditional techniques, which were the first techniques developed and popularized during the early and mid 1980s, tended to be very indirect and difficult to use safely and effectively. Although apparently easy to use and still in widespread use, these techniques often produce false identifications, and I do not recommend them. The traditional techniques are

T-1) Lexical Analysis
T-2) Carving up existing Data Flow Diagrams

*T-1) Lexical Analysis* can be used to identify objects, classes, attributes, states, and operations based on corresponding parts of speech found in problem descriptions and requirements documents. Initially developed by Russell Abbott[5] and popularized by Grady Booch,[6] this technique of equating nouns with objects and verbs with operations was quite popular during the early 1980s. However, there are many problems with this approach—as documented in my previously cited book[1]—due to the vagueness of natural language; therefore, it should only be used with great care. People are not used to writing precisely and at a consistent level of abstraction. More recent versions of this technique, using more parts of speech, may provide greater control, but these methods are much harder to use and, therefore, are less practical.

*T-2) Carving up existing Data Flow Diagrams (DFDs),* based on data stores, was a popular identification technique during the mid-1980s, when organizations were still using structured analysis as a front end for object-oriented design. However, functional decomposition often scattered the attributes and operations of objects both horizontally and vertically throughout the hierarchy of DFDs, making it difficult to rebuild the objects (i.e., the Humpty Dumpty effect), making traceability of requirements very difficult and resulting in the redundant development of partial variants of the same class by different teams working from different subsets of the DFDs.[1] Although objects encapsulate both data (e.g., attributes) and operations on that data, that does not mean that an object's attributes can be equated with a data store or that its operations can be equated with the transforms associated with the data store. Although I was one of the many methodologists who independently discovered this technique, I have long since abandoned it as unnecessary and too risky for practice.

## CONCLUSION

As you have seen, a great many techniques have been developed for the identification of objects and classes. There are also many reasons for learning more than just one or two of these techniques. Because all are in current use in the industry, engineers may well run across any of them and should be aware of their relative strengths and weaknesses. The limitations of the weaker techniques illustrate the strengths of the more advanced ones.

Learning several ways of doing the same thing offers flexibility and reality cross-checks. Some techniques are more easily learned and used than others. Different developers have different preferences. After all, a craftsman is known by his tools. When all you have is a hammer, everything begins to look like a nail.

While it is not critical that every engineer master all these techniques, master craftsmen have many tools in their toolboxes, even if they usually rely on only a small percentage of them. I have found that different engineers will use a small number of techniques that work best for them. The best engineers rely on these techniques, relegating the others to occasional usage, often as back-up techniques for verifying the results of their favorite techniques. The best engineers also know the limitations of the different techniques and do not rely on obsolete tools when better ones are available. It is not important which techniques you use as long as

**This is a major reason why objectbases often exhibit one to two orders of magnitude of performance improvements over relational databases. Using objects, one often avoids joins and paging in and out of memory

they work for you and produce good objects and classes.

With so many techniques available, identifying objects and classes should be easy. The following questions are especially interesting:

- How does one know that one has identified good objects and classes that collaborate well together?
- How does one identify larger collections of objects and classes (e.g., small increments of development such as assemblies or clusters, or patterns such as idioms, mechanisms, and frameworks) that do not clearly correlate to things in the application domain as do objects?

I will address both of these problems in future columns of ROAD. As always, I appreciate reader comment on the topics covered in these columns and will happily incorporate your recommendations in future books and articles.

### References


1. Firesmith, D. Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, John Wiley and Sons, New York, NY, 1993.
2. Firesmith, D. Whiteboards, flip charts, and JAD workshops, ROAD 1(1):44–48, May–June 1994.
3. Firesmith, D. Modeling the dynamic behavior of systems, mechanisms, and classes with scenarios, ROAD 1(2):32–36, July–August 1994.
4. Jacobson, I. *et al.* Object-oriented software engineering: A use case approach, Addison-Wesley, Reading, MA, 1992.
5. Abbott, R. Program design by informal English description, Communications of the ACM, 26(11) 1983.
6. Booch, G. Software Engineering with Ada, Benjamin/Cummings, Menlo Park, CA, 1983.


## QUALITY: IN A CLASS OF ITS OWN

*continued from page 39*

- the total quality of all the referenced component releases.††

This definition, however, requires "addition" of Quality. As was stated above, a first-cut algorithm for addition is just to take the minimum of the Quality values (this shifts the emphasis to defining the comparison operators). A better addition, however, is to take the minimum of each aspect of Quality. Thus, for example, if aspects of Quality are defined for Reviewed By (values in ascending order: none, developer, team, project) and Documentation Completeness (values in ascending order: code comments, maintenance, maintenance, and customer), the minimum of each aspect would be taken.

Development methods and company-specific development process definitions should define rules for determining the quality measures of components. When new procedures for verification and validation are introduced into the software development process—and, therefore, tools developed to facilitate these new procedures—they may be defined in terms of services on the Quality class that determine the way the quality measure is assessed. The whole set of procedures may develop in an evolutionary and controlled manner without the requirement for replacing useful tools when additional facilities are introduced. A basic framework of this kind can be introduced now with very basic quality assessment mechanisms, and it will continue to be relevant as new forms of verification and validation are supported.

### CONCLUSIONS

Quality and aspects of Quality are measurable values. Identifying a class to manage the values of Quality assigned to components encapsulates an important aspect of the software development environment that is shared across multiple tools. However, it not meaningful just to say that a particular version of a component has a particular quality value, since this also depends on the components referenced in the content of the component. The concepts of Configuration and Release are therefore introduced: respectively, they have a value for internal quality and total quality. These concepts provide a foundation for multiple tools cooperating in the verification and validation of components. From components, the complete systems that are constructed from them may also be validated.

In any software development project today, there is much more software *not* developed by the project than developed by it, including, for example: operating system, database management system, classes and algorithms supplied by the compiler, user interface libraries, code generated by development/testing tools, networking software; and so on. It is all code that will affect the quality of a delivered system. Increasingly, we can expect supported library products to extend into specific application areas—for example, classes relevant to retail financial services, medical systems, and sales management rather than just very general purpose components. Such products will allow much greater flexibility to projects than will packaged solutions, and development costs will be lowered and lead times shortened ever more dramatically. To be effective, however, it requires a consistent approach to quality assessment of components—components which are built on components. This article has begun to outline such an approach.

### References


1. Carmichael, A.R., Editor, Object Development Methods, Towards a Common Object-Oriented Meta-Model, SIGS Books, New York, 1994a, pp. 321–333.
2. Carmichael, A.R. Building tools for the verification and validation of large models, Proceedings of Object Development Experiences (Unicom Seminars), Uxbridge UK, 1994b, pp. 167–175.
3. Coad, P. and E. Yourdon. Object-Oriented Analysis, 2nd Ed., Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. Parnas, D.L. Designing software for ease of extension and contraction, IEEE Transactions on Software Engineering, SE-5(2), 1979.


Andy Carmichael, Ph.D., MBCS, C.Eng., is a specialist in software engineering methods and tools. He is a frequent speaker at national and international conferences and is editor of Object Development Methods, published by SIGS Books. As the director of Object UK, with particular responsibility for consultancy and training services, he has worked with many clients using the Coad/Yourdon approach to object-oriented development and other methods, such as HOOD and Jacobson's OOSE. His current research interests include the relating of multiple object modeling techniques to a common framework (common object-oriented meta-model) and the use of object modeling in business reengineering.

†† Since this is a recursive definition there must be some components which do not depend on any others for the rules to be valid. It also demonstrates why cyclic dependencies are undesirable (Parnas, 1979)