Semantic Models

Donald G. Firesmith

Donald G. Firesmith is a senior member of the technical staff at Knowledge Systems Corporation. He may be contacted at dfiresmith@kscary.com or 73664.3513@compuserve.com.

ALMOST ALL MAJOR object-oriented (O-O) development methods have one or more graphical models for representing static architecture in terms of classes and such important semantic relationships among them as association, inheritance, and aggregation. In this column I discuss the required properties of such models and document the latest version of the semantic model graphics of the Firesmith method.

DESIRED PROPERTIES

A good static architecture modeling technique should be able to represent all important things and static relationships between them in a clear, intuitive, and unambiguous manner. Although the most important kind of thing to represent would certainly be classes, it is also sometimes important to represent objects, terminators, and clusters. Important static relationships would include links, associations, aggregation, inheritance, and classification. Because we are modeling things and their relationships, a graphical notation of nodes (things) and arcs (relationships) would clearly help human understanding of the resulting model. A good static architecture modeling technique should be able to capture the following kinds of nodes and arcs.

Classes

A class is a definition of similar or identical objects (its instances). As such, the class is the basic building block of most designs. A class is also the unit of inheritance, whereby child classes inherit features (i.e., properties, behavior, rules) from their parent classes. An abstract class* cannot be instantiated to create a semantically complete object and should therefore not be instantiated. However, a concrete class can be instantiated to produce a semantically complete class. A root class is one that does not inherit from any other class (e.g., the class Object in Smalltalk). Generic (i.e., template) classes are parameterized and provide a powerful alternative to inheritance.

Finally, whereas most classes are sequential† (i.e., they instantiate sequential objects without any inherent threads of control) because most object-oriented programming languages (OOPLs) are sequential, other classes are concurrent (i.e., they instantiate concurrent objects with one or more inherent threads of control). These distinctions are important because sequential and concurrent objects have very different behaviors, and the graphical notation should distinguish them. As illustrated in Figure 1, the Firesmith method uses a doubled rectangle to represent sequential classes and a doubled parallelogram to represent concurrent classes. The icon is doubled to signify that a class typically has multiple instances. A parallelogram was chosen to represent the parallel processing performed by concurrent classes. Concrete classes are drawn with "concrete" solid lines, whereas abstract classes are drawn with dashed lines.

To show the components of a cluster, pattern, or mechanism, a typical static architecture diagram will contain between 3 and 15 nodes. Although many popular notations (e.g., those of Booch, Coad, Rumbaugh, Shlaer/Mellor) attempt to list some or all of the features of the class inside the class icon, this is only rarely practical. In addition to attributes and operations, there are also component parts, exceptions, invariants, etc. Most classes therefore define (and inherit) a great many features, far too many to list in an icon without the icon expanding to fill the entire page. Even when there is room to list the names of the features, what about other useful information (e.g., their signature and type)? This use of impractical icons is probably the result of using the same icon for two different purposes: (1) teaching the concept of a class as an encapsulation of features by using trivially small or incomplete examples and (2) documenting actual classes during real development. Another source of this problem has been the traditional delivery of paper documentation and the lack of adequate computer-aided software engineering (CASE) tools. Clearly, a better solution would be to use a simple icon containing only the name (and possibly a few

*A *deferred class* is an abstract class that declares one or more deferred characteristics that must be supplied by descendants before instantiation.

†I much prefer the terms *sequential* and *concurrent* over *passive* and *active* because although all sequential objects are passive (i.e., they are reactive, passively waiting for messages before performing operations), some concurrent objects are passive and use their thread only to ensure mutually exclusive access to their properties, whereas other concurrent objects are active (i.e., they are proactive and may execute operations without waiting for messages).

adornments). A CASE tool can then be used as desired to expand the icon into one or more pop-up windows containing all relevant information such as abstraction, responsibilities, properties (e.g., attributes, parts, exceptions), behavior, rules (e.g., invariants), a whitebox interaction diagram showing the interactions among the features, state model, etc.

Objects

Whereas many applications are primarily interested in classes and treat all instances of the same class identically (e.g., BankAccount), other applications may use instances of the same class in different parts of the design (e.g., Sensor, Motor). In engineering domains, it is typically important to show individual objects on static architecture diagrams, and these diagrams should therefore be called neither *class* diagrams nor *object* diagrams.

Terminators

Objects and classes often interact with and model the various direct and indirect terminators of an application. These terminators may be actual classes of real-world or system objects or they may be the roles (i.e., actors) that these external objects play. Such terminator objects may be different abstractions depending on whether the modeling is being done by systems or software engineers. The client/server relationships among the terminators are typically the opposite of those among the internal objects and classes. Static architecture diagrams should therefore provide a way to model terminators so that these different abstractions related in the opposite direction are not confused with internal objects and classes. By capturing both terminators and their associated internal objects and classes on the same diagram, the rationale for the internal objects and classes is captured and confusion between internal and external objects and classes is avoided.

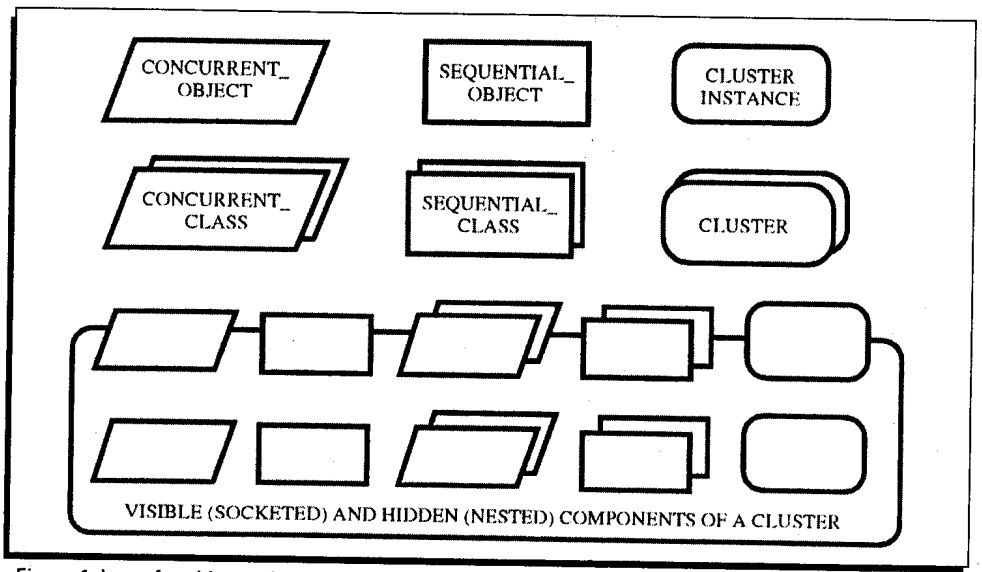

Figure 1. Icons for objects, classes, and clusters.

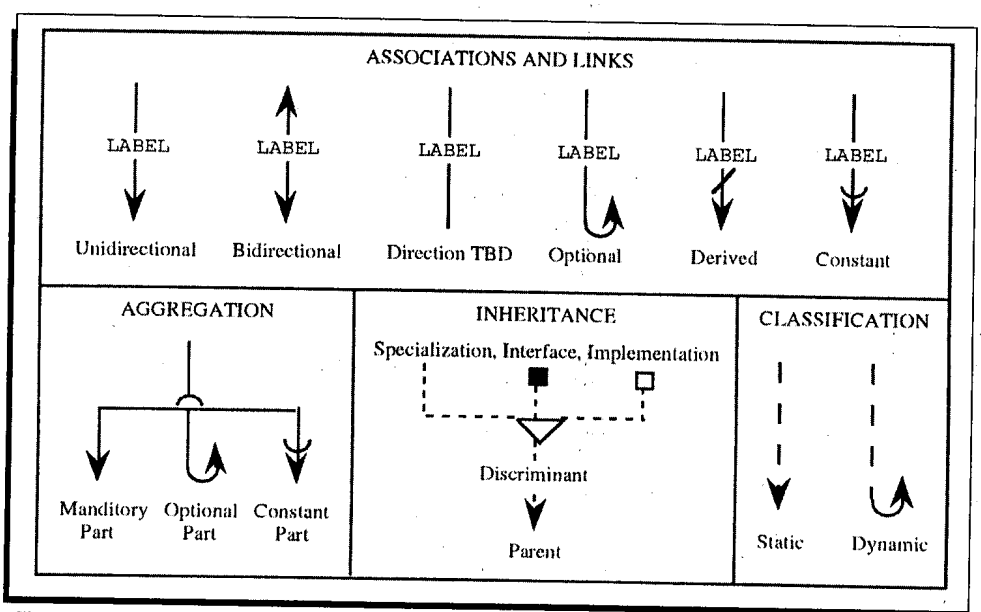

Figure 2. Icons for static relationships.

Clusters

A cluster (a.k.a., class category, subsystem, kit, ensemble) is a small group of collaborating classes that defines (i.e., can be used to instantiate) a group of collaborating instances of those classes. Clusters are usually the smallest increment of team development, being the amount of software that (1) is analyzed, designed, coded or reused, tested, and integrated by a team of two to five developers over the course of a few weeks and (2) can be documented on a typical static architecture diagram that is small enough to fit on a single page. Clusters can be defined in terms of to other clusters via cluster inheritance. Clusters support information hiding, making some of their classes visible to classes outside of the cluster while encapsulating the remainder of its classes. Like design patterns, clusters are a means of decomposing an application using building blocks bigger than a class.‡ Static architecture diagrams should support clustering, making it clear which classes are exported from the cluster and which are hidden.

Links

A link models any general named static relationship among two or more objects. Links imply visibility and thereby enable the passing of one or more messages and the propagation of any resulting exceptions. If links are viewed as roads between objects, then messages and exceptions form the traffic on these roads. As with roads, links can be either unidirectional or bidirectional. Most related objects exhibit a client/server relationship with the client referring to, sending messages to, and possibly handling exceptions raised by the server. Most links are therefore unidirectional from the client object to the server object(s). An example of a unidirectional link would be "Don is the father of

‡A cluster is typically larger than a design pattern and is often built around a few collaborating key abstractions.

Abbey." On the other hand, some objects are colleagues of each other. Such tightly coupled objects are often connected by bidirectional links that are often implemented as two related unidirectional links, each the inverse of the other. An example of a bidirectional link would be "Don and Becky are married." Referential integrity becomes an important issue when dealing with bidirectional links in order to ensure that each associated unidirectional link remains the inverse of the other. Although the direction of the links may be unknown early in the analysis, it is important that the model capture the direction early because it has an important impact on the meaning of the link and the implementation of the classes of the associated objects. For example, the link "The manager manages the employee" and the link "The employee manages the manager" have quite different meanings and would result in very different protocols for managers and employees. When three objects are connected by a chain of two links (e.g., A → B and B → C), then there often exists an implied link connecting the objects on the end of the chain (i.e., A → C). Thus links can be divided into two categories: *derived links* that are implied by the existence of other links and *base links* that cannot be derived from other links. Most links are mandatory, but some links are optional. Likewise, although some links are constants, others are variable and can be created, broken, reattached, and destroyed on the fly.

The static architecture modeling technique should include a notation for links among objects that clearly captures:

- Unidirectional, bidirectional, and direction-to-be-determined links. Although the label of the link should be consistent with the direction of the link, it should not be the only clue as to the direction of the link because labels do not always provide enough information to software developers, who are often not domain experts.
- Base vs. derived links.
- Mandatory vs. optional links.
- Constant vs. variable links.

Associations

An association is any class of links that models a general named static relationship among two or more classes. Thus, any semantic relationship that is not important enough to have its own icon (e.g., aggregation, inheritance) is modeled as an association. As classes of links, associations imply visibility and thereby enable the passing of messages and propagation of exceptions among the instances of the classes they connect. Therefore, everything that was stated previously about links also applies to associations. Because associations connect classes, the cardinality of associations (i.e., the number of instances linked) and qualifiers that restrict linkage are also important. Because associations, like links, are most often unidirectional, the traditional "crows feet" notation for cardinality (or its equivalent) should be replaced with cardinality annotations near the arrowheads on the association arcs. No separate notation is required for associations that have properties and behavior because such associations are best modeled as associative classes that are associated with the classes associated by the original association.

Aggregation

Aggregate objects contain other objects as component parts, and this whole-part relationship is sometimes important in a design. For example, a body contains organs made of tissues containing cells that in turn contain organelles made of molecules made of atoms made of subatomic particles, etc. The static architecture diagramming notation should support aggregation relationships between clusters, classes, and objects, either by nesting icons or by an aggregation relationship arc. The notation should also be able to document the cardinality of component parts and whether those parts are optional or mandatory, constant or variable.

Inheritance

Inheritance is a mechanism for creating a new definition (e.g., class) from one or more existing definitions by reusing (i.e., inheriting) parts of the existing definition(s), overriding parts of

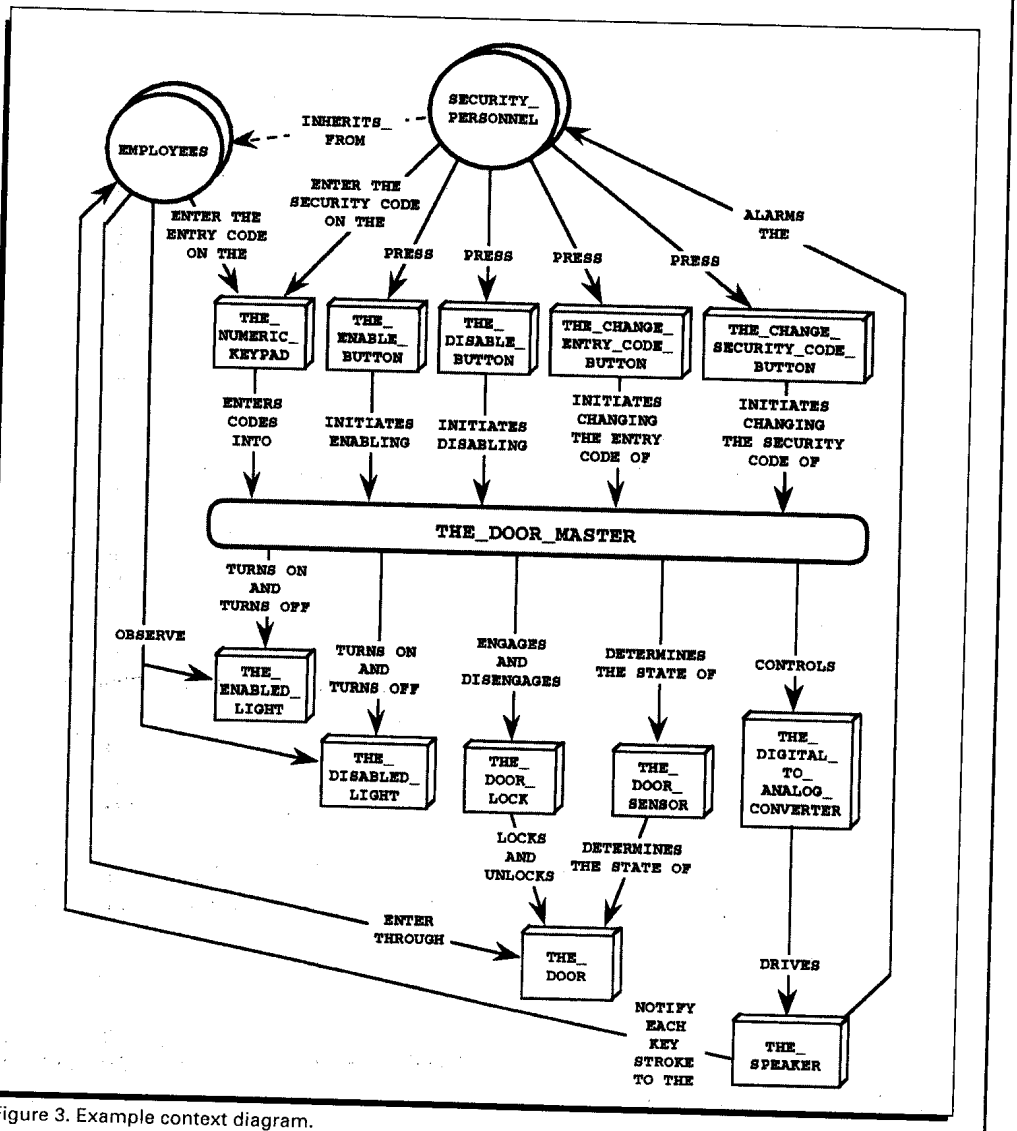

Figure 3. Example context diagram.

the inherited definition, and adding new parts to the resulting definition. Inheritance is most often used to derive child classes from their parents but can also be used to define new clusters and types. Because inheritance is an implementation technique, it can be (mis)used to relate almost any two classes. To support understandability and polymorphism, inheritance should primarily be used to implement specialization inheritance, in which the child is a specialized version of its more general parent(s). Specialization inheritance implies interface inheritance, which guarantees that the interface of the child conforms to the interface of its parents, although the converse is not true (conformity could be due to chance and not be related to the underlying abstractions of the classes). The most dangerous type of inheritance is implementation inheritance, in which a child class was created merely to obtain access to the implementation of the parent class. Implementation inheritance should typically be replaced by either aggregation or delegation (which can be represented on static architecture diagrams as associations). Because most inheritance relationships should be specializations, no annotation should be required for specialization inheritance. I recommend annotating interface (blackbox) inheritance with a blackbox and implementation (whitebox) inheritance with a whitebox.

The children of a parent may be disjoint (i.e., having no common instances) or may overlap. Similarly, the children of a parent may form a cover (i.e., each instance of the parent is an instance of at least one child). There may be discriminants, the values of which determine how the parent class is partitioned by its children. For example, the class of vehicles can be partitioned by manufacturer, by engine type, by purpose, etc. The static architecture diagramming notation should therefore be able to differentiate the different kinds of inheritance, the discriminant of a partition, and whether the child classes make up a disjoint cover, etc.

Classification

Whereas inheritance is a relationship between two definitions (e.g., classes), classification is the relationship from an instance of a definition to its definition (e.g., from an object to its class). Although it is rare that one needs to document such a relationship on static architecture diagrams, it is nevertheless sometimes useful and should be supported by a static architecture diagramming notation.

SEMANTIC NETS

Version 4.5 of the Firesmith method uses semantic nets to capture most§ views of the static architecture of an application or domain in terms of things and the important semantic relationships among them. Extended entity relationship diagrams were rejected as inadequate and because they tended to imply a traditional data modeling view rather than a responsibility-driven view of the model. Figures 1 and 2 document the icons currently used by the Firesmith method for semantic nets. Semantic nets come in the following major varieties in order to simplify the diagrams and not mix too many apples and oranges on the same diagram:

§ I also use configuration diagrams to document the overall architecture in terms of dependency relationships among clusters.

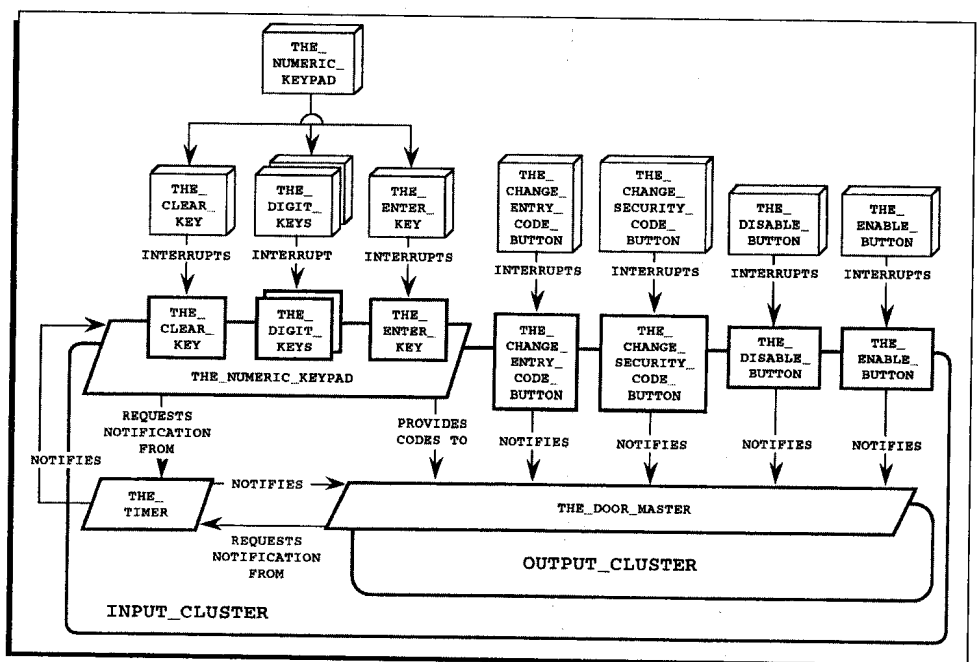

Figure 4. Example general semantic net.

Context Diagram

A context diagram (CD) is any specialized semantic net used to document a system or assembly and the links/associations between it and its primary and secondary terminators. A CD should be one of the first diagrams produced. Semantic nets have several advantages over traditional data flow diagrams when used as CDs. They are at a higher level of abstraction, easier to read, and more appropriate for use with objects. Figure 3 is an example software context diagram for the Door Master security system (for a description of the Door Master security system see Firesmith[1]).

General Semantic Net

A general semantic net (GSN) is any semantic net that documents a collection of related objects, classes, clusters, terminators, and the important semantic relationships between them. I usually develop one GSN for each cluster and pattern that I wish to document. I typically concentrate on links and associations, only documenting aggregation, inheritance, and classification relationships when it is useful for simplifying or understanding the objects and classes. For example, because child classes inherit associations from their parents, I will use inheritance relationships to clarify which classes have which associations. Figure 4 is an example general semantic net for the Input cluster of the Door Master security system.

Aggregation Diagram

An aggregation diagram (AD) is any specialized semantic net that documents all or part of an aggregation hierarchy. I only rarely find it useful to create aggregation diagrams because the incremental, iterative, parallel development cycle implies that collaboration (cluster development) and inheritance (class development) are typically far more important than aggregation. However, ADs are useful for capturing the aggregation structure of complex aggregates. Figure 5 is an exam-

ple aggregation diagram for the door object, which is provided for illustrative purposes only as the door is probably too simple to justify the creation of a separate diagram.

Inheritance Diagram

An inheritance diagram is any specialized semantic net that documents all or part of an inheritance structure showing the relevant classes, the inheritance relationships between the child classes and their parents, and where useful and practical, the generic parameters of generic classes, the interfaces and implementations of classes, their instances, and the classification relationships between instances and their classes. I sometimes use inheritance diagrams to provide a summary of part of an inheritance structure, but I much prefer to use a class browser to browse the structure to see who is inheriting what from whom and to flatten the hierarchy to see the "complete" definition of a class.

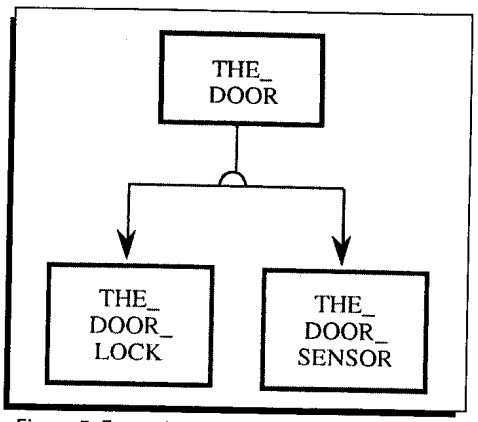

Figure 5. Example aggregation diagram.

CONCLUSION

The use of semantic nets produces diagrams that are very readable and scalable. Objects and classes are treated as software blackboxes; rather than trying to list their features on the icons, CASE tools can provide pop-up windows allowing the developer to display and update class information. Links and associations are emphasized over aggregation and inheritance, which are typically better viewed using browsers. Arrowheads should be used to naturally portray the typical client/server relationships between classes, making links and associations highly readable as normal English sentences. Because all arcs are drawn in the direction of dependency, control flow and the order of compilation and optimal testing are clear. Finally, clusters are provided to group classes and restrict their visibility.

Reference

1. Firesmith, D.G. Modeling the dynamic behavior of systems, mechanisms, and classes with scenarios, REPORT ON OBJECT ANALYSIS & DESIGN 1(2):32–36, 47, 1994.

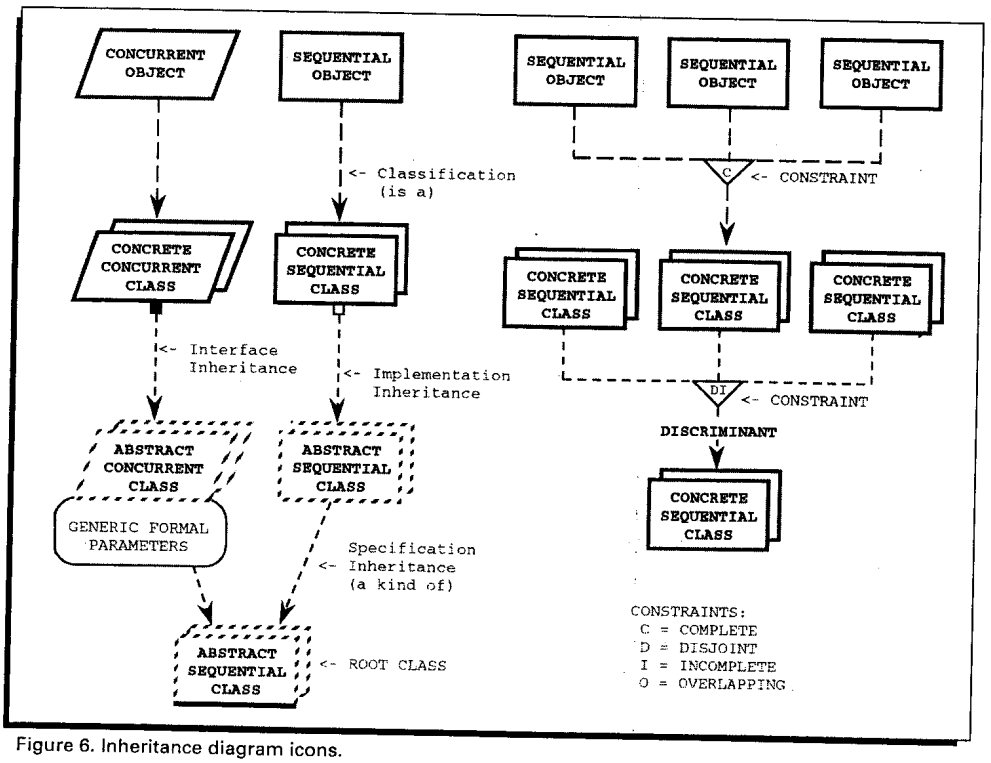

Figure 6. Inheritance diagram icons.

continued from page 12

management, TOOLS '91 TUTORIAL NOTES, ISE, Santa Barbara, CA, 1991.

18. Henderson-Sellers, B. and J.M. Edwards. The fountain model for object-oriented system development, OBJECT MAGAZINE 3(2):71–79, 1993.
19. de Champeaux, D., D. Lea, and P. Faure. OBJECT-ORIENTED SYSTEMS DEVELOPMENT, Addison-Wesley, Reading, MA, 1993.
20. Henderson-Sellers, B. Towards a process metamodel architecture: II. Analysis and design issues, REPORT ON OBJECT ANALYSIS & DESIGN 2(3):11–13,17, 1995.
21. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
22. Wirfs-Brock, R.J., B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.
23. Booch, G. OBJECT-ORIENTED DESIGN WITH APPLICATIONS, Benjamin/Cummings, Menlo Park, CA, 1991.
24. Thomsett, R. Management implications of object-oriented development, ACS NEWSLETTER, October 1990.
25. Henderson-Sellers, B. A BOOK OF OBJECT-ORIENTED KNOWLEDGE, Prentice Hall, Englewood Cliffs, NJ, 1992.
26. Booch, G. OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.
27. Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley, Reading, MA, 1992.
28. Barrett, M.L. and G.C. Simsion. A review of diagramming notations for object oriented development, TOOLS9, Potter, J. and B. Meyer, Eds., Prentice Hall, Sydney, 1992.
29. Constantine, L.L. and B. Henderson-Sellers. Notation matters: I. Framing the issues, REPORT ON OBJECT ANALYSIS & DESIGN 2(3):25–29, 1995.
30. Zhao, L. and E. Foster. ROO: Rules and object-orientation, PROCEEDINGS OF TOOLS15, Prentice Hall, Englewood Cliffs, NJ, 1994.
31. Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. Object-Oriented Development: The Fusion Method, Prentice-Hall, Englewood Cliffs, NJ, 1994.
32. Johnson, R.E. and B. Foote. Designing reusable classes, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 1(2):22–35, 1988.