Guest Column

Inheritance diagrams: Which way is up?

Donald G. Firesmith

INHERITANCE IS A fundamental part of any object-oriented architecture, regardless of whether that architecture consists of requirements, design, or code. Inheritance has many uses including organizing classes into taxonomies, facilitating reuse of both interfaces and implementations, and formatting requirements and design documentation. However, inheritance also violates localization and encapsulation by scattering the definition of a class across all of its superclasses, making it harder to know what it is inheriting and from whom, especially if multiple inheritance is used or the inheritance graphs are deep (include many levels of inheritance). Thus, understanding the inheritance architecture is critical to understanding the application.

Because each subclass* inherits from one or more superclass(es),† inheritance is a directional relationship between classes. Therefore, inheritance forms directed acyclical graphs with classes as the nodes and individual inheritance relationships as the arcs. Although code captures these relationships in textual format quite adequately for the compiler, graphics (e.g., inheritance diagrams) are a much more understandable way to document inheritance graphs for humans, and almost all object-oriented development methods use some form of inheritance diagram to document inheritance graphs.

Donald Firesmith is President of Advanced Software Technology Specialists (ASTS), a consulting and training company specializing in object technology. He is author of OBJECT-ORIENTED REQUIREMENTS ANALYSIS AND LOGICAL DESIGN: A SOFTWARE ENGINEERING APPROACH (Wiley, 1993), OBJECT-ORIENTED METHODS, STANDARDS, AND PROCEDURES (Prentice Hall, 1994) He can be reached at 4018 South Harrison, Fort Wayne, IN, 46807; 219.745.7928; f: 219.747.9389.

CURRENT PRACTICE

Figure 1 illustrates the different notations used by major object-oriented development methods for documenting inheritance relationships. These notations vary in terms of the orientation of superclasses and subclasses as well as the direction and type of the inheritance arc.

Object-oriented development methods use four primary approaches to organize classes on inheritance diagrams. By far the most popular approach places the superclasses above their subclasses. Another approach is the exact opposite, placing the superclasses below their subclasses. One method places the superclasses to the left of their subclasses, analogous to the way some developers and browsers use indented lists to document single inheritance trees. Finally, some methods nest subclasses within superclasses. Table 1 documents the superclass/subclass orientation approach chosen by major object-oriented development methods.

Four primary approaches are also used to represent the inheritance relationships on inheritance diagrams. Some methods use an arrow drawn from the superclass to the subclass, whereas others draw the arrow from the subclass to the superclass. Some methods use inheritance icons combined with either arrows or undirected line segments, whereas other methods only use a line segment with the conventions of the previous paragraph being used to differentiate subclass from superclass. Table 2 documents the type and orientation of the inheritance arc chosen by major object-oriented development methods.

All these approaches are logically equivalent and can be transformed easily into one another. The software engineering principle of uniformity strongly suggests that the object community should standardize on a single approach to inheritance diagrams to be used by all methods. I would like to see one of these approaches recommended based on the grounds of human understandability (the primary reason for the diagrams in the first place) rather than relying on merely the popularity of specific development methods or the decisions of CASE tool vendors.

* a.k.a., child class, derived class, descendent, heir, specialization, subtype.

† a.k.a., ancestor class, generalization, parent, supertype.

APPROACHES TO SUPERCLASS/ SUBCLASS ORIENTATION

First, we will consider the four primary approaches that methods use to organize super- and subclasses:

- Superclass above its subclasses.
- Superclass below its subclasses.
- Superclass to the left of its subclasses.
- Superclass contains its subclasses.

The most popular approach (i.e., placing superclasses above subclasses) has little to recommend it other than its popularity among methodologists. Although the prefix sub usually means "under, beneath, below, or lower in rank" and the prefix super usually means "over, above, on top of, or higher in rank," the words subclass and superclass are instead derived from the mathematical concepts of subset and superset. A subset is not below its superset, but rather contained within its superset. Thus, placing superclasses above their subclasses is based on popular misconception rather than valid intuitive reasons. This approach, unfortunately, inverts the inheritance tree produced by single inheritance, placing the root class in the air and the branches down in the ground. This approach

therefore relies on the rote memorization of a counterintuitive orientation and should be discarded before it becomes too much of a de facto standard to change.

The second approach places superclasses below their subclasses. This approach is very intuitive and easy to remember:

- As previously mentioned, single inheritance forms a tree structure. This approach places the *root class* in the ground at the bottom of the diagram and the branches (classes) and leaves (instances) up in the air at the top of the diagram. Like a real tree, the branches of the inheritance tree grow upwards as new subclasses are created as specializations of existing superclasses. The fact that multiple inheritance forms a graph rather than a tree is no counterargument. We still speak of root classes and leaf nodes. Also, botanists have discovered a "tree" consisting of many individual trees whose branches have grown together into a single colony that provides a good illustration of the structure of a multiple inheritance graph.
- Some methodologists and languages use the term *base class* rather than root class. A base should still be at the bottom of a structure.
- Reuse via inheritance from existing classes is a *bottom-up* activity. Inheritance diagrams should show that inheritance graphs grow bottom-up by extension, which is only possible if superclasses are drawn below subclasses.
- Subclasses are *built on top of* existing superclasses because they reuse the resources defined in their superclasses.
- Finally, the software engineering principle of uniformity implies that the style of inheritance diagrams should be consistent with that of other object-oriented diagrams. Message passing in interaction or collaboration diagrams implies a *client-server relationship* between objects and classes that, in turn, implies that objects and classes can be classified as either masters, agents, or servers.[‖] This client-server hierarchy is best illustrated when masters are placed above agents and agents placed above servants. By this reasoning, subclasses should be placed above their superclasses because they are clients of their superclasses.[#]

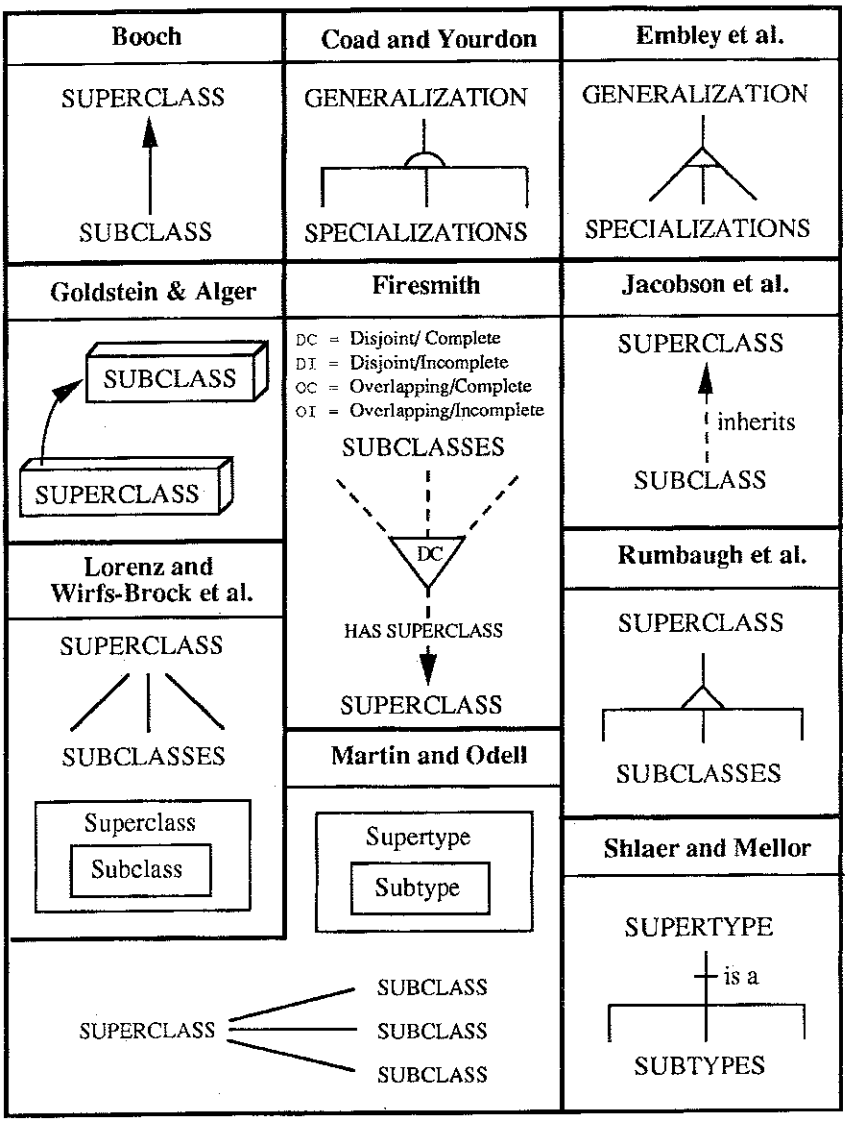

Figure I. Notations for inheritance.

Only one method (i.e., Martin and Odell) places superclasses to the left of subclasses. This approach is useful for documenting single inheritance hierarchies as ordered lists but has less value as a diagramming technique, especially because indented lists are inappropriate for multiple inheritance.

Finally, some methods nest the icons for subclasses within the icons of their superclasses. This approach is very intuitive, being based on the Venn diagrams of set theory. Unfortunately, this approach does not scale up well and is impractical for showing an inheritance tree more than two levels deep. By the time one nests an icon within a nested icon, it becomes impossible to read the label of the innermost subclass. Nesting is also impractical for dealing with complex multiple inheritance or addressing a case where the same superclass has multiple subclass structures. For these reasons, nesting should be restricted to the illustration of inheritance theory and not used as the primary mechanism for drawing inheritance diagrams.

It is clear from the preceding arguments that the best approach from a human un-

‖ Masters request services via messages, but do not receive messages. Agents both send and receive messages. Servants only receive messages.

A subclass is defined in terms of its superclasses, requires their resources, and requires visibility of them.

derstandability viewpoint is to place the subclasses above their superclasses. Because the vast majority of the software community has yet to transition to object technology, the current popularity of the opposite approach should be less important than the goal of making inheritance easier for beginners to understand.

APPROACHES TO INHERITANCE ARC ORIENTATION

Next, we will consider the four primary approaches methods used to draw inheritance relationships:

- Arrow drawn from the superclass to its subclass.
- Arrow drawn from the subclass to its superclass.
- Line segment between superclass and subclass.
- Inheritance icon combined with line segment between superclass and subclass.

Only one method (i.e., Goldstein and Alger) draws inheritance relationships as arrows from the superclass to its subclasses. Intuitively, this means that definitions and reusable resources flow from the superclass to its subclasses.

Conversely, five methods draw inheritance relationships as arrows from the subclass to its superclass. This approach recognizes that subclasses are the clients and superclasses are the servers; it is consistent with the interaction diagrams, collaboration diagrams, and dependency diagrams that draw arcs from clients to servers. This approach thus implies that subclasses depend on their superclasses, have visibility of them, and inherit from them. This approach is also consistent with all major object-oriented languages in that the subclass points to its superclass, and not the other way around.

Every method seems to have its favorite icons... the choice of icons is method-specific and probably too controversial to standardize today.

Some methods only use line segments to represent inheritance relationships. This approach is counterintuitive because inheritance is definitely a unidirectional relationship. This approach can also be ambiguous for developers who do not remember the convention of placing subclasses relative to their superclasses when the difference is unclear from the class identifiers (e.g., when the developer is unfamiliar with the application domain).

Finally, several methods use special inheritance icons (e.g., triangles, semicircles) in addition to line segments and the superclass/subclass orientation to signify inheritance and distinguish superclasses from subclasses. The primary advantage of specialized inheritance icons is that they can be labeled to document constraints on inheritance (e.g., disjoint vs. overlapping or complete vs. incomplete sets of subclasses). This approach also suffers from using line segments rather than arrows, although the existence of the inheritance icon mitigates the problem by implying direction.

It is clear from the preceding arguments that the best approach from a human understandability viewpoint is to draw the inheritance arrow from the subclass to the superclass and to use a standardized inheritance icon (e.g., triangle), where appropriate, to capture additional inheritance constraints. Optionally, interface vs. implementation inheritance can be differentiated by use of appropriate notations (e.g., open vs. solid circle, respectively) added to the inheritance arrows.

RECOMMENDATIONS

Every method seems to have its favorite icons for documenting classes and inheritance arcs. The choice of icons is method-specific and probably too controversial to standardize today. I would therefore like to concentrate on only part of the standardization problem and let the CASE vendors provide translation between method-specific icons. Based on the preceding arguments, I recommend the following industry-wide conventions for diagramming inheritance relationships:

- Draw subclasses above their superclass(es).
- Draw inheritance relationships as directed arcs (i.e., arrows) from subclasses to their superclasses.

Table 1. Superclass/subclass orientation

Superclass/ Subclass Orientation	Superclass Above Subclass	Superclass Below Subclass	Superclass Left of Subclass	Superclass Contains Subclass
Booch	Yes	No	No	No
Coad and Yourdon	Yes	No	No	No
Embley et al.	Yes	No	No	No
Firesmith	No	Yes	No	No
Fusion	Yes	No	No	No
Goldstein and Alger	No	Yes	No	No
Henderson-Sellers	Yes	No	No	No
Jacobson et al.	Yes	No	No	No
Lorenz	Yes	No	No	Yes‡
Martin and Odell	Yes	No	Yes	Yes
Rumbaugh et al.	Yes	No	No	No
Shlaer and Mellor	Yes	No	No	No
Wirfs-Brock et al.	Yes	No	No	Yes§

‡ Also uses inclusion on Collaboration Diagram.

Table 2. Inheritance arc orientation.

Inheritance Arc Orientation	Superclass to Subclass	Subclass to Superclass	Line Segment Used	Inheritance Icon Used
Booch	No	Yes	No	No
Coad and Yourdon	No	No	Yes	Yes
Embley et al.	No	No	Yes	Yes
Firesmith	No	Yes	Yes	Yes
Fusion	No	No	Yes	Yes
Goldstein and Alger	Yes	No	No	No
Henderson-Sellers	No	Yes	No	No
Jacobson et al.	No	Yes	No	No
Lorenz	No	No	Yes	No
Martin and Odell	No	Yes	No	No
Rumbaugh et al.	No	No	Yes	Yes
Shlaer and Mellor	No	No	Yes	Yes
Wirfs-Brock et al.	No	No	Yes	No

§ Also uses Venn diagrams.

- Use a similar, although different, pen style to capture classification relationships from instances to classes (e.g., because solid lines are usually used for message passing, I suggest dashed lines for inheritance and dotted lines for classification).
- Where appropriate, allow developers to graphically document via icon or annotation:
  1. Constraints on inheritance:
     - disjoint vs. overlapping subclasses
     - complete vs. incomplete sets of subclasses
  2. Abstract, concrete, deferred, generic, and root classes
  3. Differences between interface and implementation inheritance
  4. Conditional inheritance
  5. Dynamic (as well as static) inheritance

This recommended approach is illustrated in Figure 2. It offers many advantages in terms of human understandability including:

- Inheritance graphs grow bottom up due to the reuse of superclass resources.
- Subclasses are built on top of their superclasses.
- Subclasses point at their superclasses on the diagrams as in the code.
- Inheritance trees have their root classes in the ground and their branches and leaves in the air.
- Inheritance arcs are drawn *from* client to server and are thus consistent with the arcs of other diagrams (e.g., messages).
- Inheritance and classification arcs are drawn differently than other arcs (e.g., messages).
- Constraints on inheritance are captured.
- Abstract and concrete classes are differentiated.
- Interface and implementation inheritance are differentiated. ■

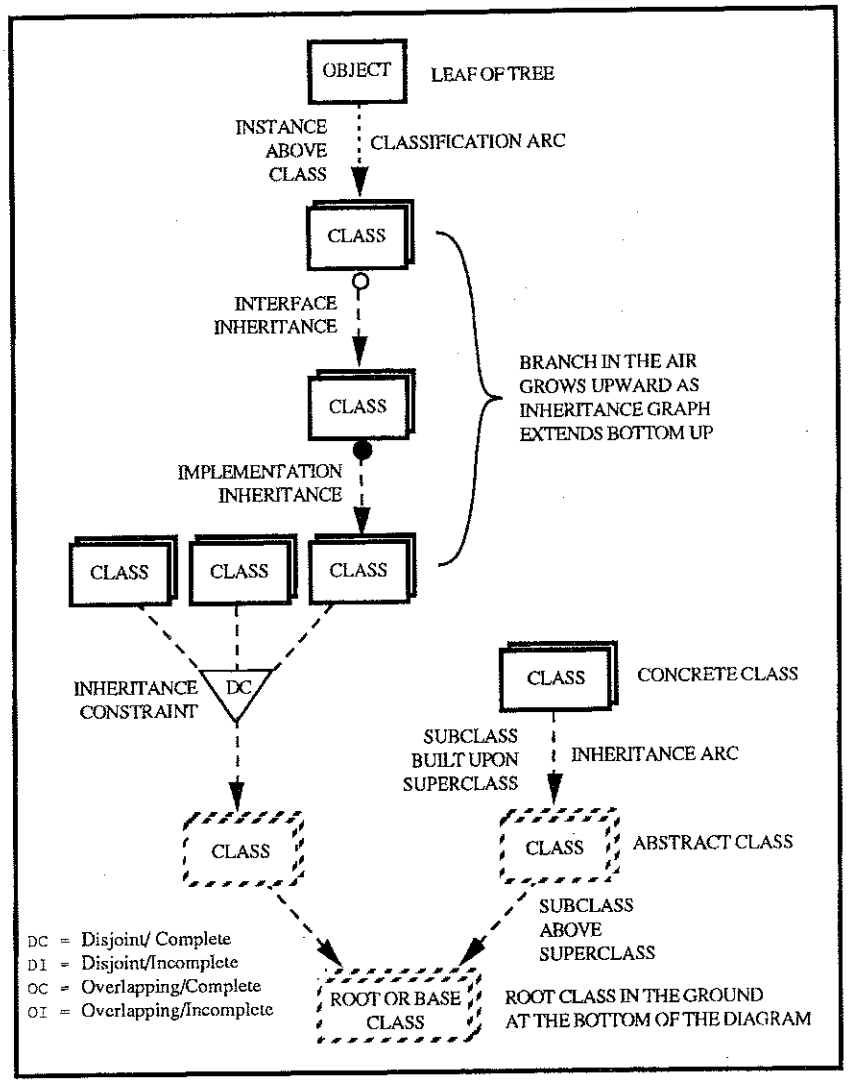

Figure 2. Recommended notation for inheritance.

References

1. Booch, G. Object-Oriented Design with Applications, Benjamin/Cummings, Menlo Park, CA, 1991.
2. Coad, P. and E. Yourdon. Object-Oriented Analysis, 2nd Ed., Prentice Hall, Englewood Cliffs, NJ, 1990.
3. Coad, P. and E. Yourdon. Object-Oriented Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
4. Derek Coleman, D. et al. Object-Oriented Development: The Fusion Method, Prentice Hall, Englewood Cliffs, NJ, 1994.
5. Embley, D.W., B.D. Kurtz, and S.N. Woodfield. Object-Oriented Systems Analysis: A Model-Driven Approach, Prentice Hall, Englewood Cliffs, NJ, 1992.
6. Firesmith, D.G. Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, Wiley, New York 1993.
7. Firesmith, D.G. Object-Oriented Methods, Standards, and Procedures, Prentice Hall, Englewood Cliffs, NJ, 1994.
8. Goldstein, N. and J. Alger. Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming, Addison-Wesley, Reading, MA, 1992.
9. Jacobson, I. et al. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokingham, UK, 1992.
10. Lorenz, M. Object-Oriented Software Development: A Practical Guide, Prentice Hall, Englewood Cliffs, NJ, 1993.
11. Martin, J. and J.J. Odell. Object-Oriented Analysis and Design, Prentice Hall, Englewood Cliffs, NJ, 1992.
12. Rumbaugh, J. et al. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
13. Shlaer, S. and S.J. Mellor. Object-Oriented Systems Analysis, Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1988.
14. Shlaer, S. and S.J. Mellor. Object Lifecycles, Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1992.
15. Wirfs-Brock, R. B. Wilkerson, and L. Wiener. Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, NJ, 1990.