Ivan Ryant

Why Inheritance Means Extra Trouble

Do squares form a subclass of rectangles? Baclawski and Indurkhya [1] explain that finding the answer is not easy. Inheritance is commonly viewed as a problem in which semantic relations mix with design concepts and the particular mechanisms of programming languages. But inheritance is more than a source of ambiguity and misunderstanding. We see another serious issue: Correct behavior of a system of objects is commonly endangered by the implementation of inheritance. Thus, the system of objects surrenders to the risk of synchronization faults that may generate nonsense data and may result in information losses, deadlocks, and so forth.

Rumbaugh says the term *generalization* refers to the relationship among class and the term *inheritance* refers to the mechanism of obtaining attributes and operations using the generalization structure. Generalization provides the means for refining a superclass into one or more subclasses. He adds that the superclass contains features common to all classes; the subclasses contain features specific to each class. Inheritance may occur at an arbitrary number of levels where each level represents one aspect of an object. An object accumulates features from each level of a generalization hierarchy [2].

What is the difference between "contains" and "inherits" relationships? The container-contents relationship links instances. Thus, the container-contents relationship requires communication among instances, but communication among their respective classes is not always necessary. Object instances may become part of a container any time, depending on their lifecycles. They may also leave the container at any time.

Inheritance links classes. It also implicitly links all the respective instances of those classes. Inheritance requires communication between ancestor and descendant classes as well as communication between ancestor and descendant instances. Does this mean that an instance of the ancestor exists for every instance of the descendant? Everybody knows an ancestor becomes part of its descendant—so a container-contents relationship exists between every descendant's and its ancestor's instance. (Embedding is a form of relationship as well as indirect association from descendant instance through descendant class and through ancestor class to ancestor instance.)

Unlike the general container-contents relationship, every pair of descendant and ancestor instances is created at once. The instances then live together (and communicate with each other) for their whole lives. Finally, they are both destroyed in the same moment. How is their common creation accomplished? Their classes create them. We can intuitively assume the descendant class creates its instance and asks the ancestor class to create its instance. In other words, the classes must communicate.

PAUL WATSON

Object is an entity or a whole. It is recognized by its behavior. No object can lose its specific way of behavior, nor can it lose its interface between its "inside" and "outside." Can an object be resolved and dispersed in another object losing its own identity? Would it be an object any longer?

What's the Matter?

Every object instance is created first. Then it interacts with its environment—it lives. Finally, the object is destroyed. That is the object lifecycle. Object lifecycle is a process or a system of several processes. Objects (or their processes) communicate with each other. Correct synchronization is necessary for any communication. That's why communication obeys rules in the form of communication protocols.

Descendant class adds new features (including new behavior) to its ancestor. But the new behavior must not violate the inherited communication protocols. This task is hard to manage when the program is created. But why?

Common implementation mechanisms of inheritance hide the communication between descendant and its ancestor. This way, some responsibility for interobject communication and synchronization is left to the software engineer. Consider three critical points:

Using inherited methods. Object behavior is usually decomposed into methods. When an object receives a message, the message is recognized and interpreted. The object may invoke an appropriate method. If the object possesses no appropriate method, it can reroute the message to another object. When the descendant receives a message and recognizes its ancestor is responsible for attending it, it hands the message over to the ancestor.

Using inherited attributes. Every attribute is an object with its own identity, memory, and behavior. Use of an attribute must obey communication protocol of both the attribute itself and the attribute's owner. The correct way to access the attribute goes through the object that possesses the attribute. Especially if the descendant instance accesses an inherited attribute, it should send a message through the ancestor instance.

Redefining dynamically bound methods. Any ancestor class may state tasks for its descendants. That means dynamically bound methods may be declared in the ancestor class but implemented in the descendant class. The ancestor states the task while also defining the communication protocol. Every request to the dynamically bound method should be routed through the ancestor instance—even if the descendant implements it.

Making Inheritance Work

These guidelines can probably help the programmer establish and keep rules of communication between ancestor and descendant instances. Implementation (especially in compiled languages) usually tends to simplify the interobject communication. In particular, ancestor and descendant processes are combined and inherited attributes and methods are shared. The programmer should be concerned with handling them as any other shared resources (mutual exclusion is required).

Generally, object lifecycles can be composed and decomposed. Any system of sequential processes may be combined, producing single sequential process (and keeping the same behavior).

Composition may be complex but the price of managing complexity should never be paid with incorrect behavior. C

Ivan Ryant (ryanti@acm.org) is a freelance software engineer and computer journalist in Prague, Czechoslovakia.

This article was formed in discussions with Petr Slaba, Jan Janecek and Karel Richta (CTU, Prague) and with Vojtech Merunka (UA, Prague).

References

1. Baclawski, K., Indurkhya, B. The Notion of Inheritance in OO Programming. *Commun. ACM 9*, 37 (Sept. 1994), 118–119.
2. Rumbaugh, J. et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1992.