Donald Firesmith

Advanced Software Technology Specialists (ASTS), 17124 Lutz Rd., Ossian, IN 46777

An expanded view of messages

MOST SOFTWARE DEVELOPERS have a strong tendency to define object-oriented concepts in terms of specific object-oriented programming languages (OOPLs) they know. Unfortunately, these definitions suffer the same limitations as the OOPLs. This, in turn, has tended to limit the power of object-oriented requirements analysis and logical design, which should be language independent. Many examples of this phenomenon exist including lack of support for dynamic and multiple classification, equating objects and classes with encapsulations of attributes and operations (which is how many OOPLs implement them) rather than as software models of application domain entities, and defining a message as a dynamically bound call to a corresponding operation (a.k.a., *method*) of an object or class. This article addresses the arbitrary, forced, one-to-one mapping of messages to operations.

Messages are the primary, but not only, mechanism by which objects and classes interact. For example, when attributes are strongly typed (e.g., as in Ada and C++) types of message parameters must also be exported to client objects and classes to provide type visibility for type checking. Constant attributes may also be safely exported under many circumstances, and exported constants more efficiently provide the same capability as exporting a corresponding preserver operation (i.e., *function*). Finally, exceptions must often be raised from server to client objects and classes to ensure proper error handling.

Many languages (e.g., Smalltalk) enforce or imply a one-to-one mapping between messages and corresponding operations. In many OOPLs, a message is defined as a dynamically bound call to an object or class that requests the performance of a corresponding operation.

During object-oriented requirements analysis and logical (i.e., language-independent) design, a message is the primary communication mechanism between objects and/or classes and may be used for the following three distinct purposes:

1. To request a corresponding service (which may or may not be implemented by a single, associated operation).
2. To provide notification that a specific event has occurred.
3. To provide data (in the form of actual parameters of the message).

We will now address each of these purposes in turn and show how they may reasonably violate the traditional one-to-one mapping between messages and operations.

MESSAGES REQUESTING SERVICES

If a message requests a service, one or more operations must surely be involved because operations are the only type of resource an object or class can use to provide services. But is only one operation per message always adequate and optimal? Consider a concurrent object running on a single processor that can simultaneously receive multiple messages. To ensure that its attributes are not corrupted due to interleaved access because of time slicing, the concurrent object must ensure mutual exclusion via critical regions (i.e., contiguous sections of code guaranteed to run to completion without interruption). All synchronous messages to such an object may well be routed to the same concurrent operation that guarantees mutually exclusive access. Some of these messages may be further routed to secondary operations that actually provide the requested services, whereas other messages may only be used to control (e.g., enable, disable) the initial routing operation. Because a one-to-one mapping does not exist between services and operations, the term *service* should not be used as a synonym for operation and messages cannot always be mapped, one-to-one, to operations. Note that this is especially common in embedded realtime applications where concurrency is a critical issue.

A similar situation occurs in applications in which operations are large and complex. This is the one situation in which functional decomposition remains useful in the object-oriented world. In this case, a single, functionally cohesive message requesting a single service may be mapped to a set of functionally decomposed operations within the same object or class.

Finally, not every service is requested by the clients of the object. When concurrent objects are used, they have a life of their own and can also perform services for their own benefit. For example, a

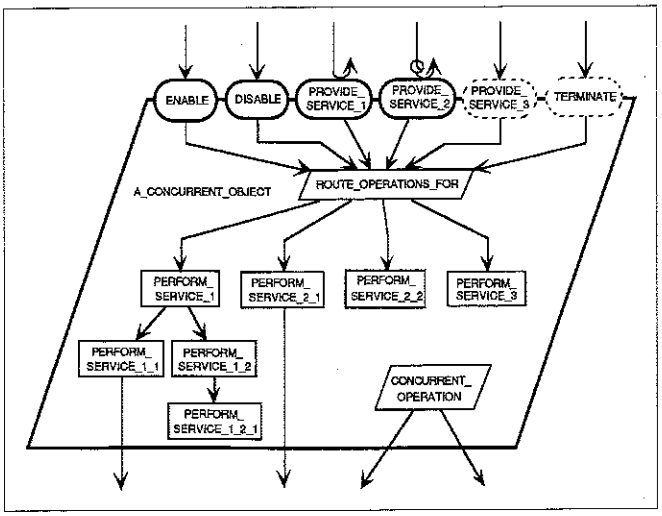

Figure 1. Example of messages requesting services.

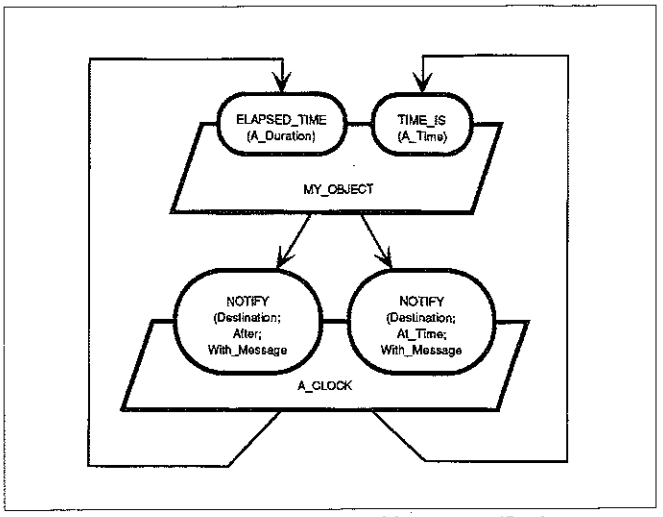

Figure 2. Example of messages used for event notification.

concurrent operation of a concurrent sensor object may cyclically poll a hardware sensor at regular intervals and update a gauge object whenever the sensor value changes a minimum amount.

Figure 1 illustrates the various aspects of this first counterexample. The concurrent object exports, and can therefore receive, the following six messages: ENABLE, DISABLE, PROVIDE_SERVICE_1, PROVIDE_SERVICE_2, PROVIDE_SERVICE_3, and TERMINATE. The first four messages are synchronous, whereas the last two are asynchronous. The third message is balking, and the fourth message is timed. All six messages are routed to the concurrent operation ROUTE_MESSAGES_FOR.A_CONCURRENT_OBJECT. The third message is further routed to the sequential operation, PERFORM_SERVICE_1, which is further decomposed into three suboperations. The fourth message is routed to two operations that provide the corresponding service. The fourth message is routed to a single operation that performs the requested service. Finally, a concurrent operation might exist that is not requested by any message, directly or indirectly. In Figure 1, six messages may be received by an object with nine operations, only eight of which are related to the messages.

MESSAGES PROVIDING EVENT NOTIFICATION

The second counterexample involves the use of messages to notify objects or classes that a specific event has occurred. For example, an object (e.g., MY_OBJECT) may need to be notified when a specific time period has elapsed or a specific time occurs. It may therefore send a "NOTIFY(Destination := MY_OBJECT; After := A_Duration; With_Message := ELAPSED_TIME (A_Duration))" message or a "NOTIFY(Destination := MY_OBJECT; At_Time := A_Time; With_Message := TIME_IS (A_Time))" message to a clock object, which must then send the appropriate message back to MY_OBJECT. These messages sent by the clock object are classic examples of messages that provide notification of events. Not only do such messages not request services, information hiding implies that the sender of the message may not even know what, if anything, the receiving object will do upon receipt of the message. This use of messages is similar to the situation in which a traveler requests a wake-up call at a hotel. The traveler may or may not perform the wake-up operation upon receipt of the message. The traveler may already be up and ignore the message. The traveler may even be using the wake-up call for a different purpose unknown to the hotel operator (e.g., a reminder to call home at a specific time), although this is typically poor design from a maintenance point of view. This counterexample also tends to be most common in the embedded real-time domain where such a message is often used for the synchronization of concurrent objects.

Figure 2 illustrates this situation. MY_OBJECT sends two overloaded synchronous NOTIFY messages to A_CLOCK, which in turn sends two messages back when the appropriate temporal events have occurred.

MESSAGES PROVIDING DATA

The third and final category of counterexamples are messages used to provide data in the form of parameters to objects and classes. As with the notification of events, these messages are often sent by objects that should not know what the receiving objects will do with the message. As before, the receiver of the messages can perform one or more operations (e.g., validate, incorporate) to deal with the data or can ignore the data (e.g., if the data is too old in a real-time system).

CONCLUSION

Most books, articles, and OOPLs assume that messages always map one-to-one to operations and so this rule is often part of the definition of "message" and "operation." This is because messages almost always map one-to-one to operations whenever only sequential objects and classes are involved. However, the situation is far more complex, particularly when concurrency is involved. Because the real-world is inherently concurrent and because concurrent objects have a fundamentally differently behavior than sequential ones, these issues must be addressed adequately during requirements analysis and logical design. Software engineers should strive to ensure that their basic concepts are not artificially limited by their tools and programming languages. ■