Determining Object Roles and Responsibilities

Reprinted from *The Smalltalk Report*
By: Rebecca J. Wirfs-Brock

"Consider the objects—books, radios, kitchen appliances, office machines, and light switches—that make up our everyday lives. Well-designed objects are easy to interpret and understand. They contain visible clues to their operation. Poorly designed objects can be difficult and frustrating to use. They provide no clues—or sometimes false clues. They trap the user and thwart the normal process of interpretation and understanding. Alas, poor design predominates. The result is a world filled with frustration, with objects that cannot be understood, with devices that lead to error."

Donald Norman
The Design of Everyday Things

I never thought I'd say this, but software objects *are* like real-world objects! Both kinds of objects are hard to use if they are poorly designed. Ensuring that software objects are easy to use involves paying attention to a number of sound design principles. No one ever said that good object-oriented design is easy. In this month's column I'll discuss the importance of understanding and modeling object roles. Once there is a clear sense of an object's intended purpose, it is much easier to detail the necessary behavior in an understandable fashion.

Identifying the central classes in an application is just the first step. Combing through a specification of the problem may provide an initial list of candidate classes, but what next? First, let me state that no designer I know has ever found all the key objects by reading and understanding a specification of the problem. A specification is just a launch pad for design activity. Depending on the weight of that specification, there will be different strategies needed to find those key classes. If there is a mound of paper to wade through, the initial task will be one of filtering out a lot of detail and focusing on identifying the highest level concepts. On the other hand, if the specification is on the slim side, the task will be to develop a skinny statement of intent into a model of key concepts that will drive the design.

There is a deceptively simple question that needs to be answered for each identified class. Can that class's purpose within the application be clearly stated? I've found it useful to force myself to write a concise, precise statement of purpose for each potential class. This purpose statement need not be long or wordy; a sentence or two will often suffice. However, if it is difficult tot construct a succinct statement; more work is needed. There are several plausible explanations

(other than that the class doesn't belong in the design) for being unable to write a clear purpose statement for a class.

1. Subdividing Large Concepts

For one thing, the class may represent too large a concept. One indicator of this is that the class seems to embody an entire program or a major portion of the overall system behavior. This large concept needs to be decomposed into more understandable pieces. What art the constituent responsibilities of this mega-object? In order to answer this question we must resolve a rather complex concept into simpler, more basic ones. These simpler concepts will be easier to understand, and their purpose and role will be easier to elaborate. Simpler concepts will be represented by classes in the final design while the larger concept may not.

It is conceivable that the large, vague concept still has a role to play and will be represented by a class in the final design. For example, the object might be responsible for coordinating the actions of other objects (each with a concisely stated purpose) that collaborate to fulfill the larger purpose. One design for an Automated Teller Machine might have an Automated Teller Session object whose purpose is to conduct a customer session. This customer session would consist of a series of user transactions with the bank (and a whole chain of responses to user requests) which are coordinated by the Automated Teller object.

Subdividing the responsibilities of a large, complex class into a number of simpler classes requires deeper understanding of the system. Each newly created class needs a clearly stated role. There already may be identified classes that can fulfill part of the responsibilities of the rather large concept. Most likely, this isn't the case. A hypothesis must then be formulated on how to partition the vague concept into several distinct roles. Each role will be assigned to a new class. A key designer of a large successful application told me that his design team subdivided responsibilities according to when, what and how. These subresponsibilities were then assigned to separate classes that were either responsible for knowing when, knowing what, or knowing how to perform an operation. Sounds simple enough. The design team found they spent time debating whether a particular responsibility was actually a 'when', a 'what' or a 'how'. One object's *what* is another object's *how*. It all depends on a particular point of view. At least the team had a strategy for elaborating class roles. But they still had to debate the details in context of their emerging model.

2. Completing a Model of Object Interactions

There are other situations where it is difficult to state a class' purpose. One common situation is that a class doesn't seem to be connected to any others. It's hard to explain why this disjoint class should exist, yet the designer remains convinced that it's important. Chances are, the class is important. The problem is that the model is incomplete. This problem typically arises when classes are sifted through one at a time, rather than building an understanding of the collaborative behavior between objects in the design.

To understand any single object's role, it must be looked at in the context of others with which it interacts. Constructing an object-oriented design is not linear, top-down process, although it is often to present the design that way. Understanding an object's purpose forces the designer to understand the roles of other objects. To understand the role of a seemingly isolated object, both an understanding of its static, structural relationships with other objects and interactions with other objects is needed.

To determine the static relationships an object has with others, examine how an object is connected to others. Is there a whole-part relationship between it and another object? Does this object represent an aggregation of other objects? If so, it is usually pretty simple to fit this object into the design.

It is much harder when an object participates in a number of relationships. In this case, it is useful to build an understanding of the dynamic behavior of the object. Performing design walkthroughs, tracing a chain of object collaborations in response to a stimulus is a good way to understand object interactions. Ivar Jacobson, pioneer of the Objectory method, introduced the notion of 'usage cases'. Usage cases can be recorded and then used to test the model under both normal and abnormal conditions. A key component of Steve Weiss and Meilir Page-Jones' object oriented software synthesis method is modeling the response to events and understanding their impacts on a design. The idea behind both techniques is to translate requirements into events, and to associate events with objects that are responsible for handling them.

The more situations that are modeled, the better. As simple as this sounds, it takes some skill to effectively elaborate object interactions. The goal is to first develop a 'big picture' before diving into detail. The way to do this is to trace object collaborations between objects that are at either the same or next conceptual level in the design. First develop an overall, high level view of key object interactions, then elaborate and subdivide roles and object responsibilities. This breadth-first approach avoids modeling classes at widely differing conceptual levels, which indeed is difficult.

This breadth-first approach represents an ideal. In practice, some areas of the design will be better understood and naturally elaborated before others. An uneven design model can make it difficult to trace object collaborations. It will be relatively easy to trace the collaborative behavior throughout the well-understood parts of the design. When collaborations are necessary with objects in an undeveloped area, suddenly what had seemed straightforward becomes very unclear. This isn't a sign of failure; it just indicates that the unclear part needs elaboration.

3. Objects that Don't Fit the Model

Perhaps one of the toughest problems to deal with is when an object doesn't fit with the designer's notion of what constitutes a 'good' object. It is very difficult to explain the purpose of such misfits. Criticisms commonly leveled against such troublesome objects are:

This is an organizing object. It is too simple. It merely consists of data. It has no behavior. Aren't objects supposed to have both?

This object's only purpose seems to be to route messages between two other objects. Why should I have intermediary between these objects? Can't they just directly communicate with each other instead?

This object is too action oriented. Aren't objects supposed to encapsulate both operations and data? This object seems like a pure 'process'. We're doing an object-oriented design, not a process decomposition.

There are no pat answers to these criticisms. In each case, the object doesn't match the designer's expectations. The model, the designer's expectations, or both need readjustment. In the first case, it is worth noting that objects are not uniform packages of operations and data. It is natural that the proportion of each will vary, according to the object's role in the design. It is perfectly reasonable for relatively simple objects to coexist alongside more complex ones. However, the object must stand on its own merit, in order to be included. Indeed, there may be preferable alternatives to creating a 'data mostly' object.

When creating an object model, the designer may need to invent mechanisms that weren't spelled out in the specification. Mechanisms may be added for the express purpose of reorganizing the flow of information and communication between objects. These mechanisms may help reduce object coupling or provide an 'abstract connection' between objects. The consequences of inserting such mechanisms needs careful consideration. But objects, whose purpose is to organize or manage communication between objects can be reasonable design additions.

In the third case, the purpose of an object may be to transform information from one form to another. Such process-oriented objects can naturally occur in a design, and are not always a sign that the designer hasn't shifted from the procedural to the object-oriented paradigm. Each process-oriented object should apply a fair amount of intelligence to produce results. Better yet, a process-oriented object can often provide a completely different view on the transformed information. The objects being processed and the clients requesting the transformed information may be only dimly aware of each other. In this case, the process-oriented object is probably a reasonable design concept. One example of a process-oriented object is a Compiler. The role of a Compiler is to transform text into a executable program structure. It takes a lot of intelligence to perform this operation. Defining a Compiler object is a reasonable design choice.

It may be that a class doesn't belong in the final design. Webster's Dictionary defines role as "a character assigned or assumed. A part played by an actor or singer." The task of the designer is to assign each object an appropriate role. Each role is constrained to fit within the existing object model, but a lot of designer discretion is still involved. It's a challenge to design well-understood, easy to use objects. But the positive impacts that well-designed objects have on application maintenance and understandability are well worth the extra effort.

References

Donald Norman, *The Design of Everyday Things*, Doubleday, 1988.

Ivar Jacobson, "Object Oriented Development in an Industrial Environment," OOPSLA '87 Conference Proceedings, Orlando, Florida. SIGPLAN Notices, vol. 22, no. 12, pp. 183-191.

Steve Weiss and Meilir Page-Jones, "Synthesis/Analysis & Synthesis/Design," proceedings of the Object-Oriented Systems Symposium, Summer 1990.