# Characterizing your Objects

Reprinted from the Feb 1992 issue of *The Smalltalk Report*
Vol. 2, No. 5
By: Rebecca J. Wirfs-Brock

In this column I'll describe some vocabulary I find useful to characterize objects. Building an application involves teamwork and cooperation. Melding classes designed by individuals into a consistent system of cooperating objects requires that team members work towards a common system architecture. Team members need to share an understanding of what constitutes well designed classes and subsystems, and what are acceptable patterns of object interactions.

Choices between perfectly acceptable alternatives must be *consistently* made across classes designed by different people. Achieving a consistent pattern of object communication first requires team members to use a common vocabulary for describing objects and their communication patterns. Once team members are talking the same language, they can have meaningful discussions about what are desirable interaction styles. Decisions can then be made based on sound engineering practices that meet business requirements.

**Stereotyping Object Roles**

Objects in our design can either be involved, active participants in many conversations, or by design play a more docile role, responding only when asked, taking a supporting role. In between these two extremes are many shades of behavior. I find t useful to classify objects according to their primary purpose as well as their modus operandi.

Here are two ways to characterize object roles:

*Business Objects* - Objects whose primary purpose is to model necessary aspects of a concept that would be a familiar to a user of the software we design. If we were designing an Automated Teller Machine for a bank, we might have Bank Customer, Bank Account and Financial Transaction objects. If we were designing an oscilloscope we might model Triggers, Waveforms or Timebases. These type of objects are also commonly referred to as domain objects since they correlate directly with concepts in the users'domain.

*Utility Objects* - These are generally useful, non-application specific objects. Smalltalk programming environments come with many generically useful classes. Classes for structuring

other objects, such as Set, Array, Dictionary, and classes representing numbers of strings fall into this category.

There are compelling reasons for application developers to create additional utility objects. On several projects I've worked on, specific individuals were assigned direct responsibility for creating, publishing, and making sure that the utility objects were appropriate to the task and properly used. It is possible to create and effectively incorporate utility objects into the application under construction throughout development and software construction.

It is extremely useful to design new utility objects that explicitly support system policies or common application programming practices. For example, we have created classes that stylize error handling and sequencing of processing steps, classes that model ranges of settable values, increments and units of measurement, and classes that monitor detectable external conditions. Once designed, these objects can be used in many places within an application.

**Stereotyping Object Behaviors**

A number of researchers and design methodologists have coined terms for describing objects according to the way they operate. My list of useful terms isn't merely a composite of all common terms in the current literature. I continue to make finer distinctions after reflecting on past experiences and tackling new design projects. My definitive list of useful behavior descriptions doesn't yet exist. This list needs periodic updating to reflect new ways of constructing software that accomplishes new tasks. Here are ways I currently find useful to classify object behavior.

*Controlling Objects* - Controlling objects are responsible for controlling a cycle of action. This cycle can either be repetitive, with conditional branching logic, or be initiated and executed once on detection of a certain set of events or circumstances. Controlling objects can initiate and control ongoing system wide activity, or iterate over a minor application task.

The original Smalltalk-80 user interface presented a stylized three-way collaboration between Model, View and Controller objects. Controller objects were responsible for responding to user directives, such as mouse clicks or keystrokes, and initiating appropriate responses. Views displayed the current state of the application, and model objects were application specific objects.

I use a broader definition than that implied by Smalltalk-80 Controller objects. Controlling objects need not be spurred to action only on behalf of user directives. Controlling objects can be found and created for many parts of an application where a cycle of activity is initiated, sequenced, and sometime later, possibly completed.

For example, in the design of an Automated Teller Machine, an ATM object can have responsibility for initializing and sequencing system interactions with a bank customer. A

further design refinement can add the concept of a Session Controller object, that controls the sequence of activities between a single bank customer wishing to carry out one or more transactions with the bank. At a lower level, there may be network controller objects responsible for handling network traffic between the application and the communication network.

*Coordinating Objects* - Coordinating objects are the traffic cops and managers within a system. Coordinators often pair client requests with desired services (or rather, objects performing a requested service). In my early object design experience, I would append Manager to the name of these objects. FontManager and StyleManager are two example class names. I used to feel uncomfortable creating objects whose primary behavior was standing around until someone needed something, then helping to establish the connection between two objects that would collaborate to actually perform some useful function. I now realize that these coordinators proved their worth simply by eliminating the need to hard-wire direct references between objects.

In another common design pattern, a coordinating object may respond to a request by briefly establishing an appropriate context, and then delegating a request to one or more objects within its sphere of influence. For example, in the Automated Teller Machine design, the Session Manager would first determine which transaction the bank customer wished to perform. The Session Manager would then create the appropriate transaction object and delegate to it the responsibility to gather additional information from the bank customer (such as amount to withdraw if it were a Withdraw Transaction) and then perform the transaction.

A coordinating object may also control a sequence of actions. It is often logical to blend coordinating and controlling functions in the same objects. A reasonable design for the Session Manager object is to give it the responsibility for creating and handling a series of bank customer transactions. A bank customer can typically perform transactions until he or she indicates a desire to terminate the session, causing our application to print a receipt of all transactions and return the customer's card.

*Structuring Objects* - Objects with structuring duties primarily maintain the relationships between application objects. In many applications, business objects have very complex structural relationships. Let's take a simplistic real world example of a file cabinet containing folders that hold documents. In this example, a file cabinet doesn't do much, it holds folders that may be tabbed and labeled. Folders simply hold their contents. It is the documents that are of interest.

In an object design, I add more or less behavior to objects in order to meet business requirements and to suit my personal tastes. I can design File Cabinets to do more than organize their contents. A File Cabinet could know when any folder was last referenced, or how much room is left in the cabinet. When I classify an object as primarily being a structuring object, I think first and foremost about what relationships it should maintain between other objects and how it

should do so, and secondarily what (if any) additional behavior might be appropriate and useful for it to have.

*Informational Objects* - Sometimes objects are created to hold values that can be asked for by many different kinds of application objects. I don't want to get into an indepth discussion of design and programming techniques to eliminate globals or minimize dependencies on hard-wired values in code. However, at times it can be useful to create objects that are responsible for yielding information. In procedural programming languages we have the ability to declare constant values. In object designs, informational objects are an equivalent concept.

*Service Objects* - A service object typically is designed to perform a single operation or activity on demand. A well designed service object provides a simple interface to a clearly defined operation. It should be simple to set up and use. Pure service objects often are the products of a highly factored design. Such a design consists of many classes of objects having highly specialized behaviors.

One reason to create service objects is to facilitate optional or configurable software features. The argument for this design strategy goes something like this: It is easier to configure a product's features by adding or removing entire classes of objects than it is to add or remove class behaviors.

As more behavior is added to a class, it can become complex to integrate new features with existing code. Optional functionality needs to be implemented in a way that guarantees pre-existing code doesn't break. Test suites and internal consistency checks become important.

When services are placed in specialized service classes, the design task shifts to one of creating an appropriate role and interface to the service object that must balance the controls the client has over the way the service is performed with simplicity and ease of use.

An operation may be so complex to perform that it warrants creating many objects to perform this service. A single object can be designed to provide the public interface to this service, hiding most of the details from the rest of the application.

Useful services can be packaged into distinct objects. These service objects might be designed so as to be useful in a variety of contexts perhaps by being easy to extend or customize. We could design our ATM transaction objects to know precisely how to print information about the transaction on a receipt. Alternatively, we could design a report object that provides printing and formatting services for the transaction object.

*Interface Objects* - Interface objects are found at the boundaries of an object-oriented application. They can be designed to support communications with users, other programs, or externally available services. Interface objects come in many sizes, shapes, flavors, and at many conceptual levels.

Interface objects can be designed to support an ongoing two way communication between some external entity. For example, in the Automated Teller Machine Application we have a number of physical devices such as the Receipt Printer, Cash Dispenser and the Card Reader. In our design, all these devices would have interface objects that define a high level interface to the services they provide. A Cash Dispenser object might define message to dispense cash, return the cash balance, or adjust the balance (as the result of either dispensing cash or adding more money to the machine).

Interface objects can be designed to translate external events or requests into messages fielded by interested application objects. For example, many external events need to be handled by the ATM system. To name a few: jamming of cash in the Cash Dispenser, failure of the door to close, the Receipt Printer running out of paper, etc. The list isn't endless, although responsible objects (most likely candidates are the appropriate interface objects) need to field those events and respond appropriately.

Or they can be designed to provide a narrow interface. For example, a menu presents a number of options and returns a user's preference. User interface objects typically support a highly stylized dialogue between the user and the system.

Interface objects are responsible for bridging between the non-object world and the object world of messages and objects. When I think about interface object design, I focus first on those objects that the rest of the applications think of as defining the interface to the outside world. I realize that many, many details can and should be encapsulated by these interface objects. The key is to hid details and provide a sufficiently abstract interface.

**Moving Object Designs along the Behavioral Continuum**

Given we have sufficiently rich vocabulary for describing object roles and behavioral patterns, we need to establish a context for applying these terms. Once we have done so, we need to evaluate our emerging design and select among alternatives. It first is useful to distinguish at what conceptual level in design should belong (as opposed to where it is currently placed). Is it a high level object, or does it provide low-level services? Does it play a significant or relatively insignificant role?

Once we determine at what conceptual level of an application an object belongs, we can easily characterize its role as a business or utility object. Examining behaviors and building cleanly defined objects takes more time. Objects don't always fall into a single behavioral category, nor do I expect them to. For instance, objects often blend behaviors of controlling and coordinating. Another common pattern is to blend behaviors for structuring and providing services into the same object.

I do find it useful to ask whether an object is assuming too much responsibility, and whether it would be more appropriate to create new classes of objects to share the load. I also distinguish

whether a design choice causes an object's behavior to shift one way or the other on a behavioral continuum. Has an object become too active or passive? Is it perhaps taking on too many behaviors by assuming both a coordinating role as well as performing a useful service? Would it simplify the design to sub-divide an object's responsibilities into smaller, simpler concepts? What would be an appropriate pattern of collaboration between that object and newly defined service objects?

When I look at rebalancing behaviors, I tend to consider the current behavior definitions for a group of collaborating objects belonging to roughly the same conceptual level. My goal is to understand and develop an appropriate distribution of control logic and responsibility among collaborators. Design creativity and individual preferences needn't be sacrificed during this assessment process. However, readjusting object behaviors needs to be purposefully done. In the next column I will discuss some object interaction styles, and strategies and reasons for choosing between them.