Adding to your Conceptual Tool Kit: What's Important about Responsibility-Driven Design?

Reprinted from the Jul/Aug 1994 issue of the *Report on Object Analysis and Design*
Vol. 1, No. 2
By: Rebecca J. Wirfs-Brock

The aspects of our model we concentrate on as we develop it, and the order in which we do so, have a profound impact on our design results. In this column I want to put a fresh perspective on responsibility-driven design and introduce you to object stereotyping. Responsibilities and collaborations are common terms in today's popular literature. Developers talk about object responsibilities when they consider what an object does. Designers describe how an object performs its tasks in terms of its collaborations with others.

We can think about an object's responsibilities at any point in our design. I think about them early. I think about them later, too. I revisit the distribution of responsibilities as I refine my design. Thinking about objects this way from the start results in clear designs. When you start describing objects in terms of their responsibilities, you tend to design objects with crisply defined roles. Responsibility-driven designs often distribute application behavior among objects differently than methods emphasizing other aspects.

Modeling Perspectives

We can describe what an object responds to (its observable behavior), its internal structures (data), how it performs its jobs (its methods), and what states it can be in and what causes it to transition between those states [Figure 1]. We can also look at an object in relation to other objects. How do others view it? How does it interact with others? How does it fit into the grand scheme of things? While all these views of an object may be important, they often don't get equal attention. In our zeal to turn our designs into code, we often visit only some of these design perspectives.

An Object?
= **Behavior** + Data ?
= **Data** + Behavior ?
= **Procedures** + Interface ?
= **States** + Transitions ?

Figure 1. Different ways to view an object

Design methods place different values and emphasis on these model aspects. Responsibility-driven design focuses on formulating an object's role before tackling its other dimensions. Design methods with a heritage from data modeling focus on data and structural relationships. Jim Rumbaugh and his colleagues view the OMT method as building upon entity-relationship modeling. Steve Shlaer and Sally Mellor emphasize the internal states of an object and what happens when an object transitions between all states. All approaches have a particular way of looking at objects.

A Horse by any other name...

Let's present an overly simplistic example to contrast various design approaches. The following example is designed to illustrate as well as entertain. Our task is to design an object that represents a horse. How *might* various methods depict a horse, an object that we all know?

The description of a data driven horse would focus in on its parts:

- head (1)
- tail (1)
- body (1)
- leg (4)

and the relationships between them. We compose a representation of a horse out of parts, each having a structural relation to each other. If we are good designers, we'd think about why we need the parts (and what we'd do with them).

A procedural or ad hoc approach, in less polite circles, starts designing the computations encapsulated by the horse. The procedural approach describes the representation of a horse by finding operations it can perform:

- walk
- run
- trot
- bite
- eat
- neigh

This definition might seem OK at first. Diving into what a horse does ignores a few important issues. Who is going to use the defined functionality and, more importantly, how they would prefer to see that functionality represented? Ad hoc approaches are a bottom up strategy that doesn't consider the system (or the clients of the objects) as a whole. As long as *you* design enough features in your horse, *I* can use it. Ad hoc methods work on a small scale, but they don't work well in the large.

An event-driven design describes the representation of a horse by first calling out the stimuli it handles, such as:

- mount rider
- dismount rider
- start
- stop
- change direction
- change speed

Once we model what a horse responds to, we need to determine how events effect the interior of our horse representation. We need to model what state changes occur for each event.

We also should think at what conceptual level we wish clients to interact with our horse representation. Which leads me to characterize a responsibility-driven horse as possibly having these roles:

- carry a rider
- run a race
- process hay into manure

Responsibility-driven design takes the stance that an object's purpose should drive its design. That's why we spend a lot of time discussing an object's role (or roles for a complex object) and how its behavior affects others around it. We can model a horse that performs all or one of these tasks, depending on our application requirements. If I were to design a horse representation that

had all these responsibilities, I'd compose it out of new objects I create, each having well-defined roles (perhaps locomotion, biological functions, such as eating and processing food).

Once we understand an object's responsibilities, we can decide the details of its public interface, how it performs its work (e.g., procedures), and what events it recognizes and responds to. We can even pin down the attributes necessary for it to do its job. Designing this way is a conscientious attempt to take a step back and taken an abstract viewpoint.

From experience, I know that when I start by designing the encapsulated data and the interface of an object, I often make conceptual errors. These errors can be insidious. I never get it right the first time. I can get bogged down in trivial details, forgetting the overall purpose of my objects. I need to force myself to back up a level. Following a responsibility-driven approach helps me think long and hard about what an object's role should be. The possibilities are endless, but what are some good role models for objects to follow?

Forming Stereotypes

A stereotype represents an oversimplified viewpoint. The American Heritage Dictionary defines a stereotype as an "oversimplified conception, opinion, or image." Objects in our design can either be involved, active participants in many situations, 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 it useful to classify objects according to their primary purpose as well as their modus operandi.

A stereotype lacks originality or creative force. Why stereotype then? Behavioral stereotypes are a useful starting point for thinking about objects. We have reasons, albeit ill-formed at first, for creating each object. We try to pick a good name, and form behavior around our ideas of how the object should perform. If we explicitly put some names to these behavioral conjectures, we can share our ideas with fellow modelers. Certain stereotypes lend themselves to considering certain behaviors over others. During exploratory modeling we test and refine our notions and consciously tune our emerging model. We start with stereotypes and enrich them by filling out model details. Our goal isn't to preserve stereotypes at all cost; instead it is to considerately craft object behaviors.

My personal list of stereotypes, as well as your list, should be added to and adjusted over time.

We distinguish whether an object serves a non-application specific purpose or if it models a concept that is specific to the problem domain. *Utility objects* are generally useful, non-application specific objects. *Business* or *domain objects* model some essential aspect of the problem domain. Business objects have some correlation with concepts familiar to our software users. We further characterize objects according to how they behave:

Controllers are responsible for overseeing execution.

Coordinators delegate tasks to others. They are a little less nosy form of controllers.

I like to distinguish controllers from coordinators using this analogy:
Controllers are more like traffic cops, while coordinators are more like old fashioned traffic lights. Both are responsible for directing traffic. Traffic cops take a much more active role. They know why they are directing traffic and can adjust their behavior accordingly. Controllers are involved in interpreting and reacting to how other objects are working. They exercise more judgment that coordinators. I can envision a traffic cop writing a ticket for a driver making an error (like bumping into another car). I can't conceive of a traffic light doing so. In fact, old-fashioned traffic lights merely switch from red to yellow to green on preset time intervals, regardless of the traffic. Coordinators may vary in intelligence. The more intelligence they have, for example, by adding sensors to the traffic light, the more they start behaving like controllers. Coordinators and controllers vary by the degree with which they take control over the objects they direct.

Structures primarily maintain relationships between other objects.

When I classify an object as primarily being a structurer, I think first and foremost about what relationships it should maintain between other objects and how it should do so. Incidentally, I consider what (if any) additional behavior might be appropriate and useful for it to have. Structurers are common stereotypes in business applications where relationships between objects can be quite complex. Some objects exist to manage these connections.

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 to meet business requirements and to suit my personal design 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.

Information Holders keep facts that other objects can ask about.

An information holder's primary reason for being is to answer questions. Continuing with our folder example, I can view a folder as primarily having the job of holding onto its contents, and stereotype a Folder as an information holder. If my concept of a Folder includes the all-important job of maintaining relationships between the items it contained, then I might nudge it towards a structuring stereotype. How I stereotype an object depends on how I view its mission from the outside, looking in.

Service Providers perform a single operation or activity on demand.

A well designed service provider presents 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. I may start out by assigning a task to one object, then discover that it is complicated enough, or reusable enough to create a service provider. I initially don't start with many service providers, but as I further refine object roles, I often create them. Service providers can be thought of as performing a task such as printing, formatting, transacting, logging, playing, displaying, compiling, calculating, computing, editing, merging, splicing, and so on.

Interfacers support communication between objects within our program and external systems or users.

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. As a further refinement, we can stereotype *database interfacers and user interfacers*. Interface objects come in many sizes, shapes, flavors and at many conceptual levels.

I don't continue to cast each design object into a single mold or stereotype for time immemorial. During initial modeling, my notion of an object's role changes and evolves. Up front exploration of alternatives is vital.

I find it useful to distinguish how 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 much responsibility by assuming both a coordinating role as well as performing a useful service? Should it let go of some control and let other objects do more of the work? Would it simplify the design to subdivide an object's responsibilities into smaller, simpler concepts? What would be an appropriate pattern of collaboration between that object and newly defined service objects?

On the Road to Object Think

When I started building object applications, my methods were long (they looked like Pascal). I used functional decomposition to structure my applications, and my objects were inflexible. The first person who had to extend my code decided to redesign it (a very smart move)! Not only did he clean up my design, he thoughtfully showed me why work was hard to use and how he re-did it. I learned a lot by studying how to rework a design.

I learned to design the interior of an object so that it could be easily extended. At one time I viewed an object as being just a bundle of methods and supporting data. I discovered that it may be more important to design how an object presents its services. If I'm focusing on reuse and extensibility, then I'll spend a lot of time working on an object's interior design once I know that it is supposed to do.

Today, I envision the task of design of an object as a packaging problem. Let me draw an analogy between the design of a cold capsule and the design of an object. A cold capsule has an opaque and a transparent part. You can only look into the transparent part.

We object designers have a much more challenging design task than cold capsule designers. Our goal when packaging an object is to support usability, clarity, and extensibility. Once we decide what behaviors to collect into a single object, we must divide them into those that are accessible to the outside and those that are not. We must present a clear interface to public behaviors. We can carefully design the interior so that subclass designers and class maintainers also have a clear picture. We might break down large methods into smaller ones, perhaps indicating those that are replaceable by subclass implementations. We can specify the parts that must be replaced, those that may be, and those that cannot. We don't merely fill up the insides of objects with code and data. Designing the interior takes hard work.

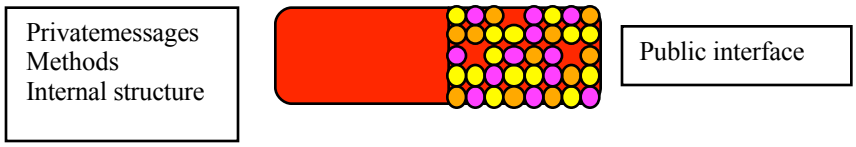

Figure 2. An object is like a cold capsule. It has a dark side, that purposefully conceals, and a transparent side that exposes a view to the outside world.

Before designing with objects, I used structured techniques. We were implementing in assembly language, so the more up front design we did, the better. We designed modules interfaces, algorithms, and data structures before writing a single line of code. All this early design work made the task of writing the code trivial. Before that, I spent several years programming in Pascal. I produced requirements documents and external design documents, then wrote lots and lots of code. We evolved the design of parts of our system many times over a period of several years. Now that I use objects, I find it even more important to design early, and to keep designing even while I'm building.

What approach should you take to designing objects? Marvin Minsky talks about the Investment Principle. As designers we need to consider that, "Our oldest ideas have unfair advantages over those that come later. The earlier we learn a skill, the more methods we can acquire for using it. Each new idea must then compete against the larger mass of skills the old ideas have accumulated."

Should you choose a design method that is most like the ways you've designed software before? Not necessarily. Work hard at being object oriented. Don't throw away all your old ideas and successful strategies to embrace this new technology. Solving problems in a new way requires hard work, practice, and positive reinforcement. Finding good objects and constructing an application out of them is a lot of hard work.

I worked hard to overcome my tendency to separate data from algorithms. I struggled to redesign big, monolithic objects into smaller, simpler, more extensible concepts. I tried to pay more attention to designing the interfaces to objects and consider coding as secondary to the ease of use. I learned not to build everything from scratch, but instead to reuse, refine, and extend where possible. I've kept good design practices, but have had to remap them to new object concepts. I still am adding new techniques to my design repertoire. In future columns I'll share more ideas about what sounds good in theory and what works in actual practice.