How Designs Differ

By: Rebecca J. Wirfs-Brock

Reprinted From:
Report on Object Analysis and Design, Vol. 1, No. 4

Most design students are searching for the 'right' set of techniques to rigidly follow in order to produce the 'correct' design. In practice, things are never that straightforward. For any given problem there are many reasonable, and a few very good solutions. We don't produce identical designs, even if we follow similar practices and design heuristics. I never cease to be amazed at the variety of designs my students produce for a given problem! For each problem we approach, we make a different set of tactical decisions. The effects of each small decision accumulates; our current design as well as our current line of reasoning shapes and limits subsequent possibilities. Given the potential impact of seemingly inconsequential decisions, we designers need to thoughtfully exercise our judgment. Design choices can only be considered in light of what we know to be relevant and important. To achieve good results, however, we need to learn to discriminate important choices from mundane ones, and to acquire a 'good' set of design techniques that we intelligently practice.

Two Boeing engineers, Bob Sharble and Sam Cohen, made a comparative study of two designs for a brewery application (1). The brewery application was fairly complex, yet could be readily designed from everyday experience. Bob and Sam didn't have to be brewmasters or process engineers to produce credible designs. In their technical report (2), they objectively measured their results using a number of metrics and presented their design models in quite a lot of detail. What a wealth of material to examine! An examination of these two designs and how they differ is the focus of this article.

What's interesting about their results is that each knew and practiced a different design method. One followed the responsibility-driven design method, the other a slightly modified version of the Shlaer/Mellor method. Responsibility-driven design emphasizes modeling object behavior; data-driven methods like Shlaer/Mellor emphasize modeling object data and state. These two very different methods naturally result in two very different paths of reasoning, and two different designs. In their report, they formed some conclusions about how these design practices influenced their results. I encourage you to read their report if you are interested.

However, methodological influences and their relative merits *are not* the topic of this article. In fact, I've seen radically different design decisions made by practitioners of any design method. This leads me to conclude that past experiences, overall design strategies, and tactical decisions have as much influence on a designer as does following any particular design method. Indeed, I want to turn my attention to exploring where and exactly how these two designs diverged, and what decisions might have caused these significant differences. I'll also speculate on what might be natural places to make and revisit these critical design decisions.

A Statement of the Brewery Design Problem

The brewery application manages physical inventory, a library of beer recipes, and production of beer. The production system supervises the creation, movement and bottling of beer. Beer is made according to a recipe which specifies ingredients and amounts. Ingredients are mixed in a mixing vat to create a batch of beer. After a batch is mixed, it is moved to a fermenting vat, where readings of its specific gravity will be taken daily. While fermenting, beer temperature will be monitored and kept within a tolerance of the temperature as specified in a recipe. After it has finished fermenting, beer is moved to another vat to settle, then moved again to a bottling vat where it is bottled on an external bottling line. The brewery includes a network of interconnected pipes and vats, an inventory of ingredients, a collection of recipes, and batches of beer both in production and already bottled.

The Boeing designers developed their designs for an "essential" brewery system. They made a number of simplifying assumptions. They assumed a single process solution with no asynchronous error handling or processing. They also purposefully chose to not consider user interface, operating system or performance issues. They restricted their designs to use only behaviors modeled by their design objects or available in number, boolean, character and string primitive data types.

Comparing Classes and Responsibilities

Names for objects representing physical concepts in the brewery were similar in both designs, e.g. Vat, Batch, Valve, Recipe, Ingredient. In fact, at first glance, even classes that managed the movement of batches between vats had similar names. Schedules Transfer and Transfer Order. Yet how tasks were accomplished was very different. The scenario for transferring a batch between vats clearly illustrates these differences.

In figure 1, we see how work and information were distributed between objects for the first, data-driven design, and the second responsibility-driven one.[1] Scheduled Transfer is responsible for determining where and how to transfer a batch. Through a series of requests, it directs valves to open and shut, causing a batch to flow through a series of pipes from one vat to another. In this design, Scheduled Transfer did most of the work. Other objects held states that were queried and manipulated by Scheduled Transfer, as it opened and closed valves and moved the batch from one container to another.

[1] I'll continue to refer to each design by number, where practical, to avoid the longer name. This is also the order in which they were written about in the report.

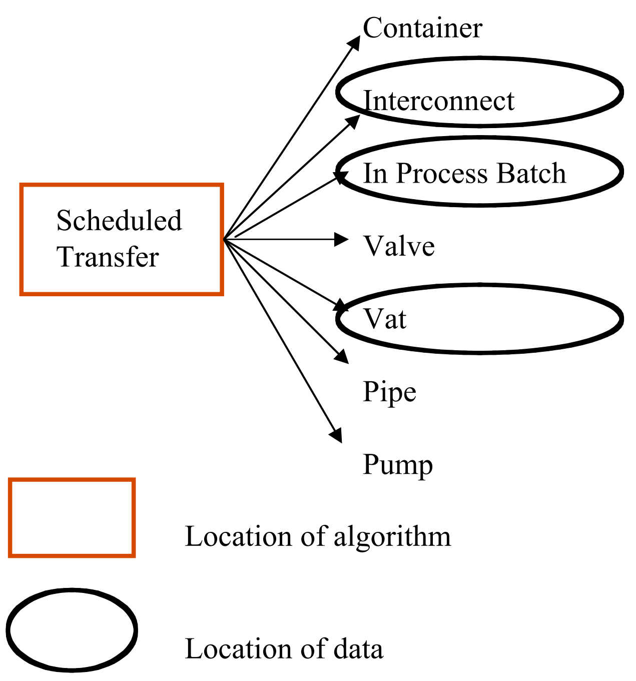

Figure 1. Transferring a Batch in the Data-Driven Design: Key Classes and Behavior
In the figure objects that were used by another have an arrow pointing at them.
Objects enclosed in a rectangle performed work, objects whose state was set or queried are encircled.

In the second design, Transfer Order shared the work of transferring a Batch with Manifold and Valve objects (see figure 2). Work and intelligence was distributed among three classes rather than concentrated in a single one.

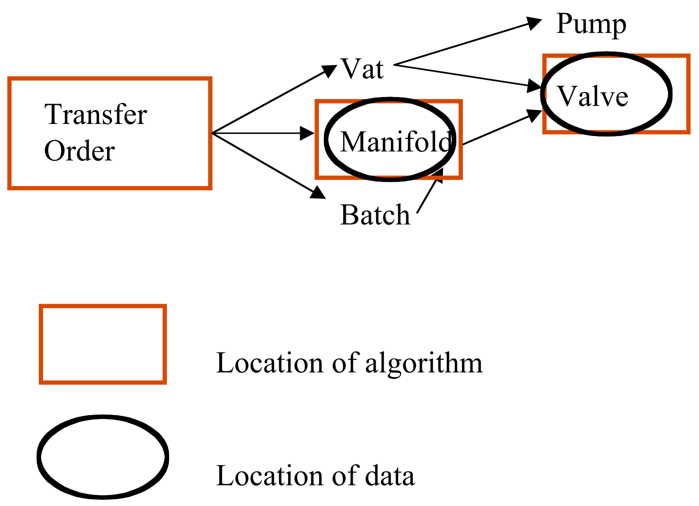

Figure 2. Transferring a Batch in the Responsibility-Driven Design: Key Classes and Behaviors

Let’s take a closer look at some key objects to see how the work was accomplished. Valves were part of both designs. Both versions of Valve understand how to set their closed or open status. In the second design, Valves are also able to form connections between Pipes and Vats. This enabled Pipes and Vats to collaborate with Valves to connect and disconnect paths between them. In the first design, valves only maintain their status, in the second they also make connections.

In the first design, Pipes and Vats were classified as Containers. Here is the specification of the Container class:

> **Abstract Class:** Container
> **Purpose:** A container is an enclosed volume that is intended to hold liquid.
> **Subclasses:** Pipe, Vat
> **Uses:** List
> **Messages:**
> set to clean, dirty or in use
> set distance and selected valve
> answer activity status questions

In the second design, Pipes and Vats were classified as Manifolds. Manifold defines responsibilities for knowing and maintaining connections between other kinds of Manifold objects. Here is a specification of the Manifold class:

> **Abstract Class:** Manifold
> **Purpose:** A container for beer that can have multiple connections
> to other containers via valves.
> **Subclasses:** Pipe, Vat
> **Uses:** Valve
> **Messages:**
> open path between manifolds, close and reset path
> determine distance between manifolds
> set to clean, dirty or in use
> add valve

In the second design, Pipes and Vats inherit the ability to open paths and determine distances between other Pipes and Vats. These objects also know how they were connected, and are tasked with calculating distances, and opening and closing pathways. In my estimation, they are fairly clever abstractions. Their behavior isn't directly inferable from reasoning about how their real world counterparts might work.

Conceiving of extraordinary behaviors is an acquired design skill. It is an inherently difficult one to teach, since how one might make objects more clever varies from situation to situation. One teacher told me how he explained the idea of smart design objects using the metaphor of 'object-oriented glasses.' We designers should adopt a perspective that isn't limited to copying the physical world; indeed our challenge is to enhance our objects' latent capabilities. We should view our objects as being inherently smarter than their real world counterparts. In fact, real world examples can be somewhat limiting. Putting on 'object-oriented glasses' lets us suspend everyday laws of physics and envision our objects' true potentials.

Consider an example of a file cabinet which contains folders that hold documents. In this real world example, the file cabinet provides some structure for its contents, but it doesn't have very interesting responsibilities. It might be viewed quite simply as holding folders that may be tabbed and labeled. Folders simply hold their contents; it is the contents of documents that are of interest.

In contrast, in an object design we can embellish our objects with responsibilities to suit our personal design tastes. We can design software File Cabinet objects to do more than organize their contents. A File Cabinet object could know when any folder was last referenced, how much room is left in the cabinet, or even maintain a history of who looked through its contents. The possibilities are only limited by our imagination and our application requirements.

In the second design, Pipes, Vats and Valves take on non-obvious behaviors that could also reasonably be performed by others (such as calculating distances and closing pathways). Combining behavior with appropriate data made them interesting. They also don't fit into a single, simple behavioral stereotype; these objects both know and do things. The first design had objects that matched single stereotypes. For example, control was concentrated into a single object, Scheduled Transfer, and information was stored in the objects it collaborated with. None of these information holders exhibited any interesting behavior, they only were responsible for knowing their current state. Table 1 summarizes the behavioral stereotypes (5) for these and some other objects.

Responsibility-Driven Class	Stereotypes	Data-Driven Class	Stereotypes
Transfer Order	Coordinator	Scheduled Transfer	Controller
Manifold	Structurer Service Provider	Container	Information Holder
Vat	Structurer Information Holder	Vat	Service Provider
Valve	Information Holder Service Provider	Valve and Interconnect	Information Holder
Batch	Information Holder	In Process Batch	Information Holder
Pump	Service Provider	Pump	Service Provider
Brewery	Coordinator	Brewmaster	Controller
Pump	Service Provider	Pump	Service Provider
Pipe	Structurer Information Holder	Pipe	Information Holder

Responsibility-Driven Class	Stereotypes	Data-Driven Class	Stereotypes
Transfer Order	Coordinator	Scheduled Transfer	Controller
Manifold	Sturcturer Service Provider	Container	Information Holder
Vat	Stucturer Information Holder	Vat	Service Provider
Valve	Information Holder Service Provider	Valve and Interconnect	Information Holder
Batch	Information Holder	In Process Batch	Information Holder
Pump	Service Provider	Pump	Service Provider
Brewery	Coordinator	Brewmaster	Controller
Pump	Service Provider	Pump	Service Provider
Pipe	Structurer Information Holder	Pipe	Information Holder

Table 1. Corresponding Classes in the Two Designs

Not only did Pipes and Vats in the second design exhibit interesting behavior, they inherited this behavior from an abstract class. When we build an inheritance hierarchy, we classify objects according to how they might commonly behave or to their shared attributes. It is considered good design practice to not subclass simply to inherit attributes or behavior. Ideally we should use inheritance to classify our objects, not just to get reuse. Yet our classification strategy, whether it is a considered blend of behaviors and attributes or not, colors our resulting objects. In the first design, Container was defined in terms of common attributes. No interesting behaviors were defined for container. This is a natural consequence if we initially define an object's data, and don't take time to consider potential behaviors. In the second design, Manifold defined common behavior and attributes.

Inheritance hierarchies are not there in the real world, waiting for use to capture and mirror them in our designs. Instead, we must create reasonable classifications according to some sense of design aesthetics. In fact, concepts in the real world do not easily map to neat, tidy inheritance hierarchies. It is our job as designers to abstract common behaviors and construct reasonable and appropriate inheritance hierarchies. This difficult task requires experimentation and reflection. As Backlawski and Indurkhya state (4), "there are no 'standard' conceptual hierarchies. Given a domain and a specific purpose, certain concept hierarchies would be clearly preferable...how inheritance is to be incorporated in a specific system...[is] a policy decision." It's up to use to classify our objects. How we classify objects and use inheritance is an important design and programming consideration.

Quantifiable Design Measures

Let's look at how these designs compared using some quantifiable measures. In our emerging field, there are few quantifiable measures that have been written about and even fewer that are useful. These following measures are one way to consider an overall design. The individual numbers for each measure aren't important. It is examining relative differences between two sets of measures that leads to further insights. Chidamber and Kermerer (3) proposed these six basic measures that were made:

Weighted Methods per Class (WMC) is a measure of the complexity of a class given by the complexity of its attributes and methods.

Depth of Inheritance (DIT) for a class is defined as the number of its ancestor classes.

Number of Children (NOC) is the number of its immediate subclasses. This is a measure of the potential influence a class can have on the design of other classes.

Coupling between Objects (CBO) is a measure of how coupled a class is to other classes not related through inheritance. CBO for a class is the number of classes that a class uses.

Response For a Class (RFC) is the number of its own and other methods that it invokes. This measures the complexity of a class by counting method invocations.

Lack of Cohesion in Methods (LCOM) is a measure of the degree of similarity among methods. In theory, methods are more similar if they operate on the same attributes.

In addition to these six measures, they also measured **Weighted Attributes per Class** (WAC), the number of attributes for a class, weighted by their size.

Figure 3 summarizes the measures for each design. It is interesting to note that Responses For a Class (RFC) is roughly twice as high in the first design, and that weighted methods per class (WMC) and weighted attributes per class (WAC) are also significantly higher. In the first design, there are more attributes, more methods and nearly twice as much message traffic.

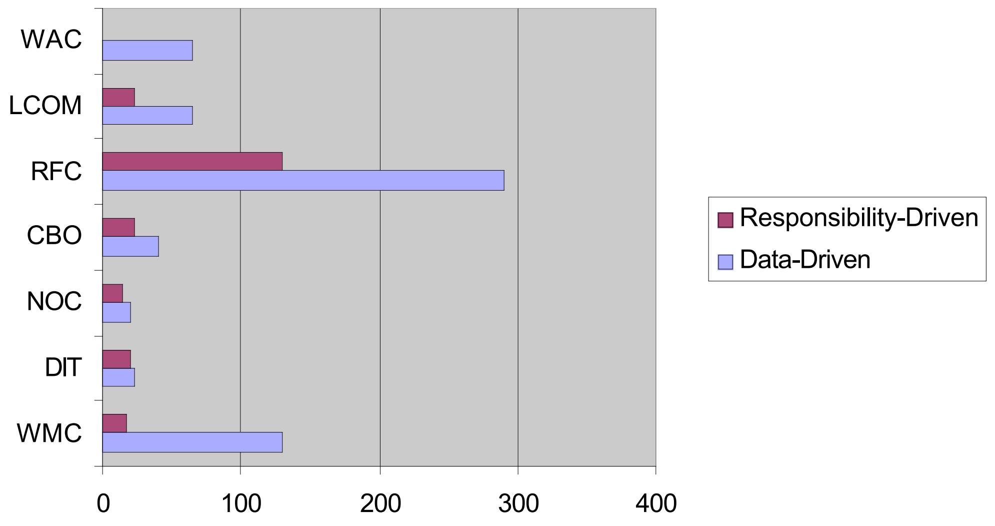

Figure 3. Design Metrics

If we look at figure 4, we can clearly see that for every scenario, there is a significant difference between the number of required messages.

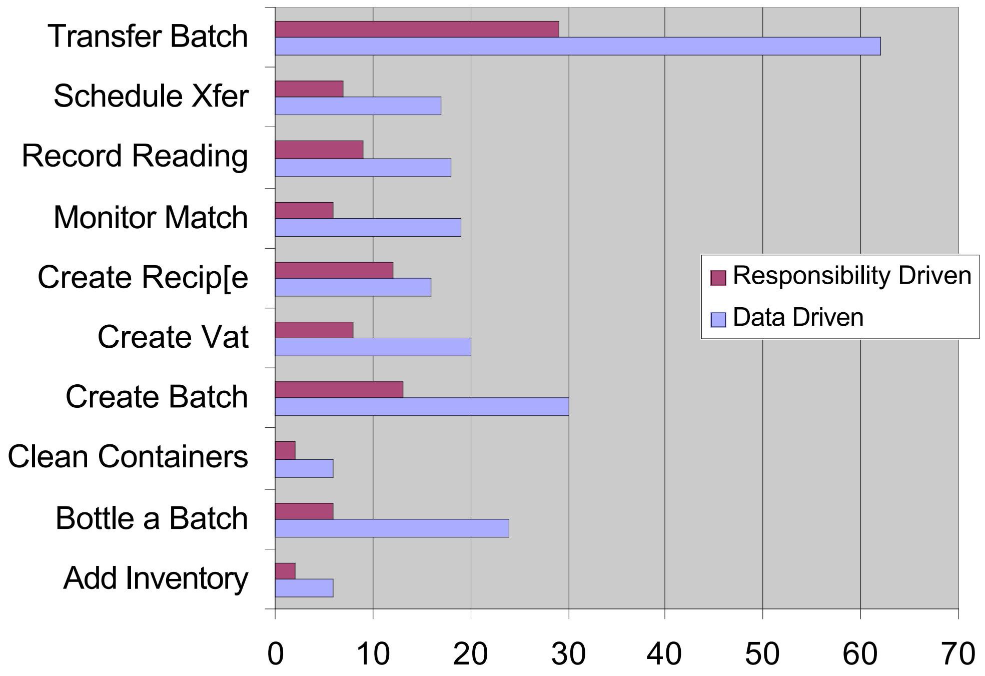

Figure 4. Messages Per Scenario

Why is this so? In the first design, objects primarily fall into one of two stereotypes: controllers and information holders. In fact, the Brewmaster and Scheduled Transfer (both controlling objects) contribute to nearly half the Responses For a Class (RFC) total. They are sending lots of messages and know about many other objects (14 and 12 respectively). In the second design, the top contender for the Responses For a Class measure is Batch (an information holder), followed by Scheduled Transfer (a coordinator) and Brewery (another coordinator). No one class had more than a 25 score. In the second design major tasks are distributed among several cooperating objects. This consistent pattern of delegation, along with higher level communications between objects contributed to most of these differences.

Characterizing the Two Designs

The above measures combines with individual class roles and responsibilities and detailed object interaction diagrams provides a pretty complete picture of each design. The important differences between these designs are summarized as follows:

Responsibility-driven design	**Data-driven design**
delegated control style	centralized control architecture
a few ‘smart’ objects that blend stereotypes	lots of simple ‘information holders’
‘coordinators’ rather than ‘controllers’	controllers setting and querying information holders

fewer messages	more low level messages
inherited behavior	inherited attributes

These designs were different, but did these differences matter? They bother were reasonable designs. However. the design metrics indicate that the data-driven design was more complex, and had more dependencies between objects. Arguably, the second design is, by some measures, better than the first.

Designing you r application's communication and control architecture is one of the most important decisions you can make. The control architecture and level of communication between controllers and the objects they controlled had a major impact. Consciously choosing an application control strategy is clearly an important decision. Deciding on a delegated control results in control and coordination tasks being placed in more objects.

We need to pay attention to those design choices that can cause us to separate data from behavior, and speculate about what object *might* do with what they *already know*. Considering object roles and responsibilities is an early tactic that can be revisited as we model more of our design.

These design strategies aren't prohibited by any particular design method. However, modeling data, then the processes to work on that data, can lead designers to place data and behavior in separate objects. These tendencies can be countered by making conscious efforts to blend behavior with data. This is our design challenge.

References

1. R. Sharble and S. Cohen, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," pp. 60-73, Software Engineering Notes, vol. 18 (2).

2. R. Sharble and S. Cohen "the Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," Boeing Technical Report no. BC2-G4059, October, 1992.

3. S. Chidamber and C. Kermerer, "Towards a Metrics Suite for Object Oriented Design," OOPSLA '91 Conference Proceedings, SIGPLAN Notices 26(11), pp. 197-211.

4. K. Baclawski and B. Indurkhya, "The Notion of Inheritance in Object-Oriented Programming", pp. 118-119, Communications of the ACM, vol. 37 (9).

5. R. Wirfs-Brock, "Stereotyping: A Technique for Characterizing Objects and their Interactions," Object Magazine, Nov/Dec. 1993.