# Characterizing Your Application's Control Style

Reprinted from the Sept/Oct 1994 issue of the *Report on Object Analysis and Design*
Vol. 1, No. 3
By: Rebecca J. Wirfs-Brock

Assigning responsibilities and determining collaborations are intimately related. Determining object interactions can be either a directed activity, or a fuzzy, ill-defined one. We need to actively design and reexamine patterns of collaboration between our objects if we want to achieve consistent design results. Before we can design our application's control architecture, we need to know our options. This article discusses several different application control styles. Some ways of distributing object behaviors and designing control styles are clearly preferable. In many cases, however, tradeoffs must be made and the winning alternative isn't always obvious.

## Looking for Control

We task certain objects with prodding others to do the detail work. Details include holding onto facts, performing calculations, interfacing to other systems, and modeling relevant aspects of our business. As a starting point, we stereotype objects as controllers, coordinators, information holders, structures, or interfaces. These objects can be generally useful, application specific, or useful in several business specific applications. When we architect our application's control style, we need to focus our attention on those objects who *coordinate, initiate* or *monitor* major application activities. All objects aren't equally important.

Objects that are active, vital centers of influence warrant more of our attention. Often these objects can be identified in an existing design by name. Having 'Manager' or 'Controller' or 'Handler' or 'Driver' in your name is a dead giveaway. Many other design objects are passive; they are important but not vital parts of a design. They are designed primarily to be used by others, and don't shape the interaction patterns of others. They have little impact on other design elements and often play a role of information holder or simple service provider.

## Characteristic Control Styles

Control and coordination tasks are often placed in multiple objects. Here are some characteristic strategies for control and coordination placement:

1. **GUI Centered** - Control in User Interface objects. Instead of merely triggering application activity, GUI objects do a major share of the application's workload.
2. **All Over the Place** - Control and coordination randomly found in many different object stereotypes. The system looks as if it were capriciously hacked together rather than

designed. The application is characterized by an inconsistent strategy that is hard to unravel.

3. **Business Centered** - Control and coordination placed in objects modeling business specific concepts. No objects found with pure coordination or controlling roles.
4. **Centralized** - Control found in a few (maybe even one) distinct objects. Other objects having small responsibilities are directly manipulated by controlling objects.
5. **Delegated** - Control carefully placed in distinct objects. Consistent patterns of delegation. Some objects were explicitly designed to initiate and coordinate, rather then directly perform large tasks.

**Taking A Closer Look**

I've encountered each of the above control strategies (or non-strategies in the case of the All-Over-the-Place style) in working systems. Each control style has its set of challenges. Before discounting the first two strategies as 'bad', and others as 'better', let's discuss the pros and cons of each style, and see how we might with the best of intentions arrive at a 'poor' design.

**GUI Centered Control**

GUI Centered designs are characterized by user interface objects that perform most or all of the application's work. We find user interface components doing much more than displaying or capturing user provided information. For example, we may find code in Button objects to directly handle button clicked events. User input may be validated and processed by objects that hold and display the information. Typically, in GUI centered designs, there are few if any objects explicitly tasked with handling the control and sequencing of the major application tasks.

Carried to the extreme, we find interface objects talking to other interfaces with no 'middleware' model of business objects whatsoever. Values are held in primitive user interface elements such as text fields or list boxes. Information that needs to be manipulated is scraped off fundamental screen elements and directly manipulated by other interface objects. Sums can be calculated and databases updated without any notion of business objects. I used to discount GUI centered designs as quick hacks that evolved out of rapid prototyping experiments and were never appropriate. The emergence of powerful visual programming environments with high powered interface components have slightly altered my opinions.

GUI centered design can work for simple applications that don't require modeling of business concepts. Hypercard applications are a perfect example of a simple yet powerful direct manipulation environment. If what you need is a tool, pick it up and gladly use it; don't waste time modeling the contents of a toolbox!

Reasonably structured procedural code can also be wrapped into an object and reused. Development environments that support 'automatic' objectification of existing functionality provide a lot of leverage. User interface and other fundamental components can interface to this 'objectified' functionality. Small to medium sized applications can successfully be developed in this fashion. While modern toolkits make this an attractive option, there are a number of pitfalls to avoid. Her are some considerations:

- There will be no reusable model of business objects after you are finished.
- Whenever the user interface or control logic changes, the control logic needs to be reconstructed and 'attached' to new interface elements. This can be tedious and error-prone.
- Whenever you add significant functionality to a user interface object, you miss an opportunity to create a reusable, non-visual object. Potentially reuseful behaviors are mixed in with specific user interface components, making them hard to find and reuse in other situations.
- Business policies can become buried in obscure places making them hard to maintain.
- Evolving GUI centered applications can be tricky. What worked before may not scale to larger amounts of information or more complex processing requirements.
- Consistent application behavior is hard to enforce among multiple developers.

## Dealing with Control found All Over the Place

Disperse, seemingly ad hoc control strategies are never planned, they just happen. Control found in unlikely places can be the result of a design that evolved over the lifetime of several developers and several application revisions. Most complex applications are carved into major subsystems before much, if any, design work is attempted. Subsystem designers end up fleshing out the interior for their subsystems. These interiors are rightfully universes of their own. Random control placement results from zealous attempts at making every object have interesting behavior. Designers may naively believe that all objects had better be active and responsible, or they haven't done a good job of distributing responsibilities!

Carried to extremes, this design strategy is fraught with problems:

- When control is spread into far reaching and unpredictable corners of an application, there are no established patterns to follow.
- Reuse of business logic is difficult.
- Understanding the implications of a proposed change or additions can be even more difficult than for a GUI centered application.
- Debugging odd side effects caused by unanticipated conditions can be extremely difficult.

While the root causes of random control placement may be different than GUI centered control, many of the negatives found in GUI centered control also apply here. When control is found all over, we have spaghetti object interactions, which are just as hard to maintain as spaghetti code was in traditional designs.

## Business Centered Control

A business centered control strategy places all the control in business domain objects. These objects represent business concepts. What's missing in this type of design are any objects explicitly tasked with modeling business processes. In general, it is a good design principle to encapsulate behavior with related information and therefore reasonable to find business objects that both know and do things based upon what they know.

Problems crop up, however, when bundling of information and behavior is carried to an extreme. When there are no objects that are responsible for managing or initiating business tasks, complex behaviors can be blended into a individual object. These overburdened, wonderfully complex megaobjects are hard to understand, maintain, and generalize for reuse. Domain wizards often can handle more complexity than those that follow and maintain their implementations. For them complexity buildup isn't intentional, it just happens. What starts out as a simple, well-defined object grows into a large complex one that may be difficult to maintain. Implementing a responsibility may be simple or quite complex. Misinformation and optimism cause us to believe a solution is simple when in fact it requires significant work.

Another even more insidious design problem occurs when control and coordination tasks are arbitrarily placed within business objects. The design problems encountered with this type of architecture are similar to all-over-the-place control architectures, even though the designers had good intentions. In fact, they probably have painstakingly tried to separate user interface details from business logic. It's just that they had no conscious strategy for modeling complex behaviors.

I've encountered this phenomenon when designers have developed a rich domain model of the structural relationships between objects before having any notion of how they should behave. Tacking on behaviors to manipulate complex structures, in this situation, seems capricious. Since any object can be navigated to from any other, placement of control and coordination tasks in any or all objects seems reasonable, but contrived. Replicating behaviors everywhere clearly seems impractical and redundant. In these situations designers end up reluctantly placing behaviors in 'core' business objects. Finding those centers can be difficult.

Introducing object stereotypes into this design situation can have big payoffs. Designers can experiment with creating new types of objects. They can design new service providers that manipulate information. This information might be assembled by a coordinating object. Tasks may be initiated by objects having a coordination role. The size and complexity of any task determines whether new service providers are useful, or whether a responsibility could reasonably be added to a core business object.

Factoring of control and behavior goes more smoothly whenever there is a natural 'hierarchy' of relationships between business objects. However, this isn't always the case. Detailed strategies for developing these solutions are beyond the scope of this article. The general notion is to build classes that model processes that manipulate complex networks of business objects, rather than embedding complex behaviors in those objects themselves.

## Centralized Control

A centralized control style is characterized by single points of control interacting with many simple objects.  The intelligent object typically serves as the main point of control, while others it uses behave much like traditional data structures.  To me, centralized control feels like a 'procedural solution' cloaked in objects (see Figure 1).

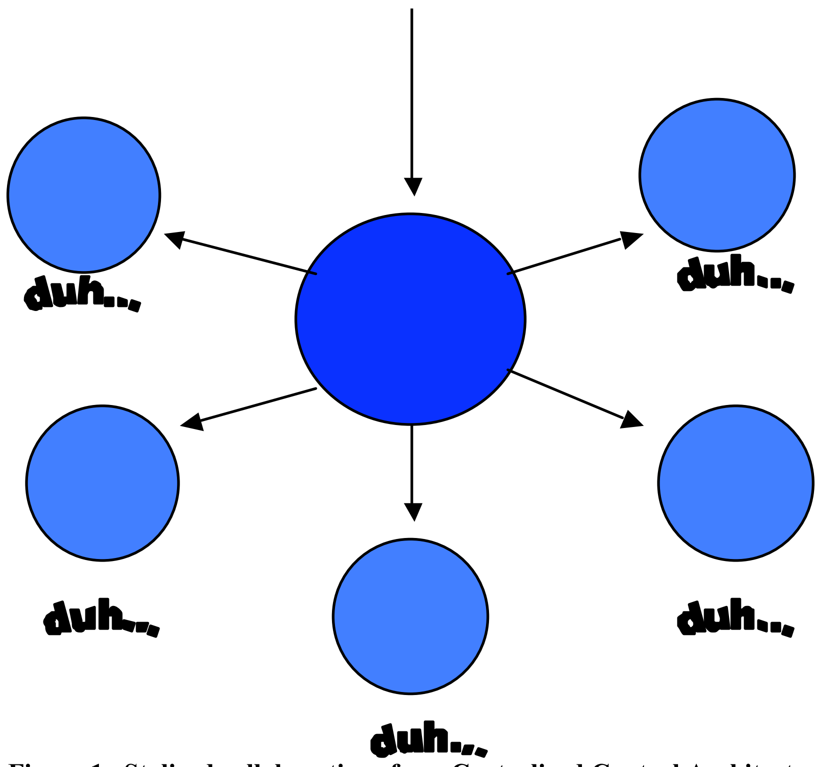

**Figure 1. Stylized collaborations for a Centralized Control Architecture**

Proponents of centralized control argue for these benefits:

- Control is easy to locate. Changing the order of task steps is much simpler if all control is concentrated into a single object.
- A simple 'recipe' for producing an object design can be followed. One controller can be created for each business scenario or use case.
- Each object surrounding a controller can be simply designed to hold facts or perform simple operations. These simple objects can potentially be reused in a wider range of applications than if they did more work.

On the other hand, there are a number of potential negatives:

- Centralized control can result in extremely complex control logic. The interior of a controlling object may be difficult to modify for fear of breaking it.
- It takes more 'simple' objects to build a solution than an architecture having more 'intelligent' objects.
- Simple objects tend to have low level, non-abstract interfaces. This can precipitate 'low bandwith' communications between controllers and the objects they manipulate. Not only does such a design require more simple objects, it requires more messaging traffic to accomplish meaningful work.
- Centralized control makes it harder to assign meaningful design and implementation work to teams. Developers are divided into the few elite who work on intricate control design, and the masses tasked with building simple objects. Those not working on controllers often don't have an accurate picture of how their objects are intended to be used. This can cause errors when developing even 'simple' objects.

**Delegated Control**

A delegated style ideally has clusters of well defined responsibilities distributed among a number of objects. A hierarchy of control may be evident. Objects in a delegated control architecture tend to coordinate rather than dominate. Tasks may be initiated by a coordinator, but the real work is performed by others. These worker objects tend to both 'know' and 'do' things. They may even be smart enough to determine what they need to know, rather than being plugged with values via external control. To me, a delegated control architecture feels like object design at its best (see Figure 2).

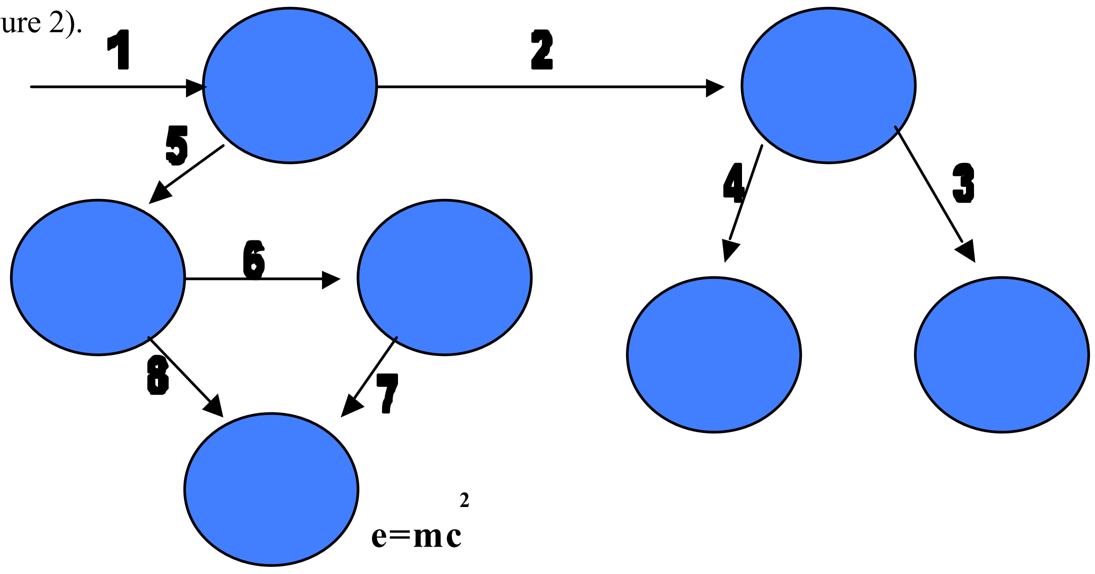

**Figure 2. Stylized collaborations for a Delegated Control Architecture**

Delegating means that coordination tasks will be spread among several objects. As a consequence, delegated control architectures can:

- require more study to understand how control and information pass between objects
- make it harder to find the details. This is particularly true when chains of delegation are long and unobvious. Objects in such a long chain may appear to 'pass the buck' rather than contribute any meaningful work (see Figure 3).
- require finer distinctions of roles and responsibilities. For example, one object may be responsible for *initiating* a task, another may *perform* it. While an object may be able to *answer a question (e.g. know a fact)*, it may do so by asking another object that is responsible for *knowing* that fact.

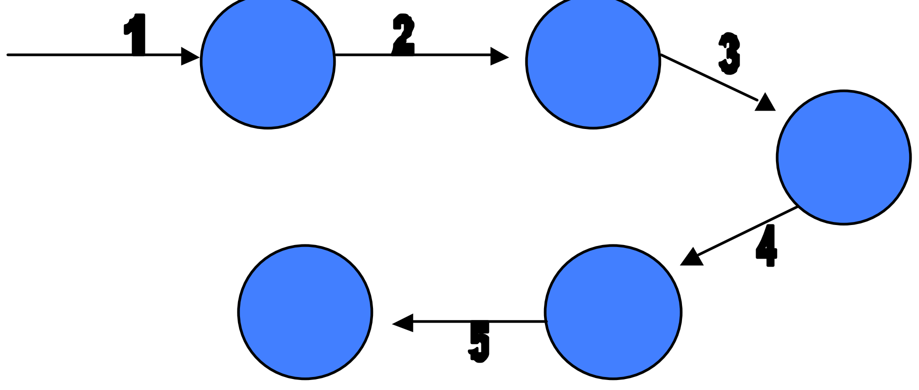

**Figure 3. Decentralized Control Carried to Extremes**

There are a lot of positive aspects to a well-factored delegated control architecture:

- Objects with coordination responsibilities generally know about fewer objects than dominating controllers. Their knowledge can be fairly shallow. Delegation means letting go of details. Coordinators don't require deep knowledge of the inner workings of those with which they collaborate.
- Changes are typically localized and simpler to make. Tasks have been factored into a number of objects having less complicated interiors.
- It can be easier to get the big picture and comprehend roles of key objects. The control strategy can be understood by understanding a few key collaborations.
- It is easier to parcel out larger chunks of meaningful detailed design work to a team once collaborations between key objects have been designed.

## How our Ideas Change

The nature of design is iterative. We make our best guesses based on imperfect knowledge and current working assumptions. We work with our model, adding details and filling in the gaps in our design model. Experienced designers expect to revisit initial assumptions as their design progresses. Novices should get comfortable with this process. Each tactical decision we make impacts future solutions. Only as we add details do the implications of our design choices become apparent. In light of our deeper understanding we discover that we should refactor object roles and collaborations.

Revisiting and fixing up a design is a sign of progress, of filling in gaps in our knowledge. It is not a sign of failure. Taking the time to consider, rather than plowing on with ill-formed strategies, is a hallmark of seasoned designers. If we adapt our objects throughout design and construction, we'll have a better chance of building in desirable characteristics.

An admirable aspect of many seasoned object developers I know is their intolerance for complexity. They refuse to let an object grow too long winded! Throughout design and implementation, they seek to simplify. They frequently refactor design objects during implementation, spawning new objects with well-defined, simpler roles. They build new service providers and create new abstractions to hold meaningful implementation information. They seem to refactor effortlessly. In actual fact, they mull over refactoring ideas as a background task while busily spending most of their time building and implementing their current design. When their ideas finally gel, they consciously fix up the design rather than go with the current flow.

Designers often refactor behaviors when they need to, not according to any prescribed plan. A student in a recent object design class told me that whenever a CRC card had too many responsibilities, it triggered him to look for ways of delegating control. Several well-defined centers of control worked their way into his design over a two day period. He couldn't force this process. It took time and background mental activity. In this fairly short gestation period, however, he came up with several meaningful abstractions.

Our knowledge will always be incomplete and our schedules always tighter than we would like. Proposed design changes must be considered in light of the time they would take to implement, the impact hey have on others, and their perceived benefits. In future columns I'll explore some aspects of this constant balancing act. I'll explore fundamental ways our designs can change and evolve, and present tactical guidance for shifting behaviors between objects.