Button, Button, Whose got the Button?

(Patterns for breaking client/server relationships)
By Robert C. Martin

Introduction

How many design patterns does it take to turn on a table lamp? This question was explored in some depth on the net recently. Someone proposed a simple application based upon a table lamp and then provided a design that he felt was reusable. There followed many critiques and counter designs and arguments. Many people submitted their own notions of how the design should look, and many others quarreled over the fundamentals of object oriented modeling.

This article represents my interpretation of that fluid and dynamic conversation. I will not mention any names, nor attempt to chronicle the actual net articles.[1] Instead I will try to present the issues and design patterns in a way that is instructive and entertaining.

The Application Description

The application that we are discussing really is just a table lamp. The lamp has a button and a lightbulb. When the user pushes the button on the lamp to the "on" position, the lightbulb goes on. When the user pushes the button to the "off" position, the lightbulb goes off. The software we are designing is the software that resides within the table lamp itself. Somehow the fact that the user has pressed the button is communicated to the software. The software must then send the appropriate signal to the lightbulb. The goal is to design this software so that it is constructed from reusable components.

At first glance, it is hard to believe that we could have a meaningful discussion about such a simple problem. Yet this problem, simple as it is, is a classic client-server scenario. Much of power of OOD is manifested in managing the dependencies between clients and servers. Thus, while the algorithms employed in solving this problem are utterly trivial, the patterns of dependency are worthy of study.

Managing Client and Server Dependencies

The table lamp problem is an extremely simple example of a Client/Server relationship. A Client/Server relationship exists between two objects if one sends a message to the other, expecting that other to perform some kind of service operation. The Client is the object that sends the message, and the Server is the object that receives the message and preforms the service.

In traditional procedural programming, clients had to depend upon servers. Servers were often implemented as functions that the clients had to link with and make calls to. Thus, the clients had direct knowledge of the server, and the source code of the client had specific knowledge of the source code of the server.

---

1. Those interested should procure the thread named "Do OO People Really Understand Software Reuse" from an archive of the comp.object newsgroup for Jan/Feb 95.

In OOD we often find it desirable to break the dependency of the client upon the server. This allows the client to be used with many different servers, so long as they all conform to the same interface. It also isolates the client from the server such that if the source code of the server is changed, the client does not need to be changed or recompiled in turn. This prevents changes from propagating up to clients from servers.

Solving the Problem

We can begin to solve the table lamp problem by examining the two use cases.

- When the user pushes the button to the “on” position, the system turns the lightbulb on.
- When the user pushes the button to the “off” position, the system turns the lightbulb off.

These use cases translate nicely into the simple object diagrams in Figure 1 and Figure 2. In each case, the **`Button`** object receives some stimulus from the outside world

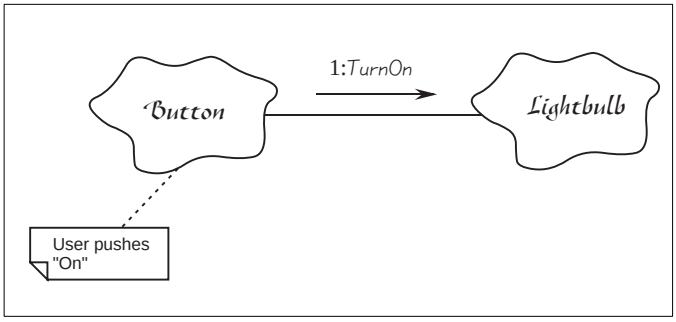

FIGURE 1.
User pushes button “on”

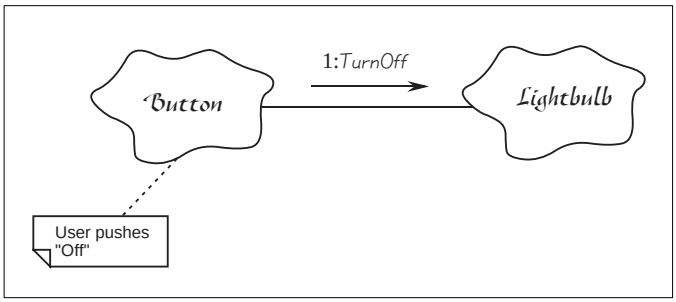

FIGURE 2.
User Pushes Button “Off”

and then sends the appropriate message to the **`Lightbulb`** object.

These two objects were chosen in order to separate the software that detects the state of the button, from the software that controls the state of the lightbulb. I felt that this separation was important because these two concepts could be reused in very different contexts. That is, buttons could be used to control devices other than lightbulbs, and lightbulbs could be controlled by objects other than buttons.

This rather simple dynamic model is supported by the static model that is shown in Figure 3. The class **`Button`** contains the class **`Lightbulb`** This "contains" relationship results from the fact that instances of **`Button`** must know what object to send the **`TurnOn`** and **`TurnOff`** messages to. The most common mechanism for satisfying this need is for **`Button`** to contain **`Lightbulb`** in an instance variable.

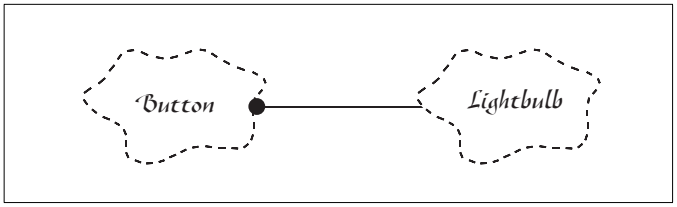

FIGURE 3.
Button contains Lightbulb

The use of the "Contains" relationship as shown in Figure 3 spawned a lot of controversy. Several people felt that the object model should reflect the real world relationships between a button and a lightbulb. Since, in the real world, a button does not contain a lightbulb, these people felt that the relationship was somehow incorrect.

However, containment in OOD is not the same as physical containment in the real world. Aggregation of objects *may* represent real world physical containment; but more often it represents some other kind of physical connection. As Booch says: "...aggregation need not require physical containment..."[1]

In Figure 3 I am using aggregation to represent the fact that there is a permanent cause/effect relationship between a particular button and a particular lightbulb. If one insists upon a real-world counterpart for this relationship, then I suppose the wire that connects a physical button to a physical lightbulb can serve. However, in my opinion, far too much effort is expended in trying to find real-world justifications for classes and relationships. As Jacobson says: "We do not believe that the best (most stable) systems are built by *only* using objects that correspond to real-life entities..."[2]

Making the Button Reusable

The disadvantage of the solution in Figure 3 should be clear. While the **`Lightbulb`** class is nicely reusable, the **`Button`** class is not. Were this design to be implemented, then the source code of **`Button`** would depend upon the source code of **`Lightbulb`**, and so **`Button`** could not be reused without dragging **`Lightbulb`** along.

This dependence of the client (**`Button`**) upon the server (**`Lightbulb`**) is the hallmark of procedural design, rather than object-oriented design. In procedural design, clients depend upon their servers, and the servers are independent. In object-oriented design, abstract polymorphic interfaces are used to break the dependence of the server upon the client.

1. *Object Oriented Analysis and Design with Applications*, 2d. ed. Grady Booch, Benjamin Cummings, 1994, p. 129
2. *Object Oriented Software Engineering*, Ivar Jacobson, Addison Wesley, 1992, p. 133

Obviously, **Button** is the kind of class that we would like to reuse. We would like to be able to control devices other than **Lightbulbs** with a **Button**. Thus, somehow, we have to break the dependency between **Button** and **Lightbulb**. There are several ways to do this, each employing a different design pattern. We will look at two popular mechanisms.

The Intelligent Children Pattern

The first solution is one which employs a pattern that I call INTELLIGENT CHILDREN. The static model for this pattern can be seen in Figure 4. The idea is that **Button** is an abstract class that knows nothing about **Lightbulb**. A special derivative of **Button**, called **LightbulbButton** knows about **Lightbulb**. Thus, **Button** can be reused in other applications to control other devices, by deriving classes from **Button** that know about those devices.

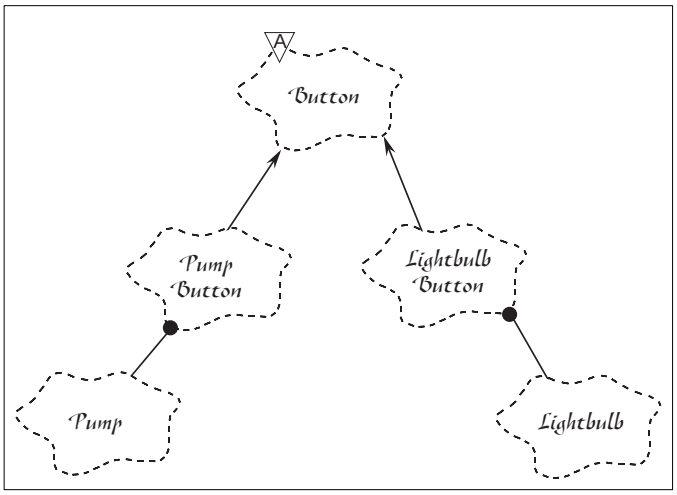

FIGURE 4.
Intelligent Children Pattern - Static Model

How does this pattern operate? The dynamic model is shown in Figure 5. The **Button** class is an abstract base that knows how to detect when a user has pushed the button. It then sends itself a message, either **TurnOnDevice** or **TurnOffDevice**. Since **Button** is abstract, it must have a derivative. If the derivative is a **LightbulbButton**, then upon receipt of **TurnOnDevice** it sends the **TurnOn** message to the contained **Lightbulb**. Likewise, upon reception of the **TurnOffDevice** message, it sends the **TurnOff** message to the contained **Lightbulb**.

Note that the **Button** class is now reusable, and that new kinds of **Button** objects that reuse **Button** can be created at any time. **Button** is reusable because it no longer depends upon **Lightbulb**. We have broken that dependency by converting **Button** into an abstract class, and supplying it with polymorphic interfaces.

This management of client server dependencies, using abstract polymorphic interfaces, is the hallmark of a good object-oriented design. It should be clear that much of the challenge in object-oriented design lies in the identification and specification of these interfaces, and ensuring that they can operate polymorphically. Generally, this information is not found by studying a model of the real-world. Instead, it is found by examining the dependencies that form as the software design evolves and then finding ways to use abstraction to break those dependencies .

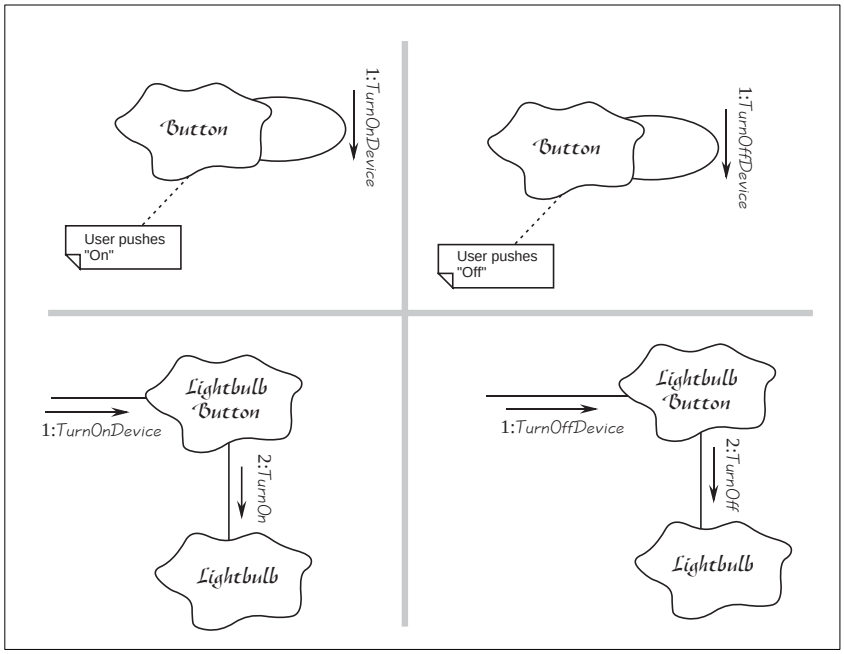

FIGURE 5.
Intelligent Children Pattern - Dynamic Model

The INTELLIGENT CHILDREN pattern is most useful when you want to reuse a class (like **`Button`**) with other classes that already exist (like **`Lightbulb`**). New derivatives of **`Button`** can be created at any time and adapted to other classes.

The Abstract Server Pattern

Another pattern that is useful for breaking dependencies, is one that is related to a pattern that Gamma[1] calls STRATEGY. I call this particular variant ABSTRACT SERVER. Figure 6 shows the static structure of this pattern. **`Button`** is a concrete class that contains a reference to the abstract base class **`ButtonServer`**. Figure 7 and Figure 8 show that when **`Button`** receives stimuli from the outside world, it translates that stimuli into **`TurnOn`** and **`TurnOff`** messages that it sends to the **`ButtonServer`**.

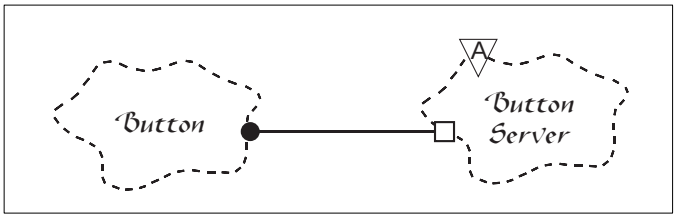

FIGURE 6.
Abstract Server static model

Figure 9 shows how the ABSTRACTSERVER pattern can be integrated with the **`Lightbulb`** class. **`Lightbulb`** is made to derive from **`ButtonServer`**. Clearly this plan is only useful if the **`Lightbulb`** class does not already exist.

1. *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, et. al, Addison Wesley, 1995

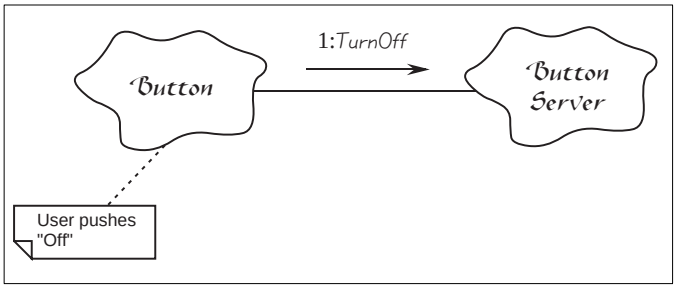

FIGURE 7.
Abstract Server Turns Off

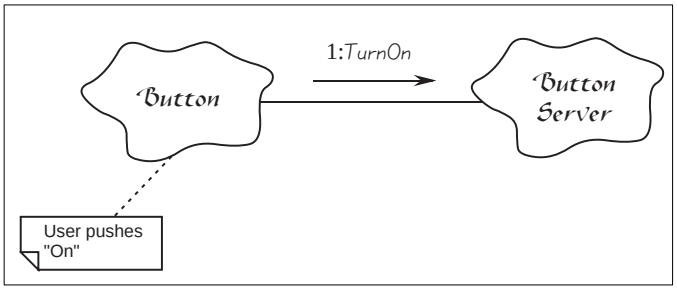

FIGURE 8.

Although this technique is very common, it has some disadvantages. Notice that it forces the **`Lightbulb`** class to have a dependency on something that is related to **`Button`**. In Booch's method, **`Button`** and **`ButtonServer`** would almost certainly belong to the same class category[1]. And so **`Lightbulb`** would depend upon the category that contained **`Button`**. This dependency is probably not desirable since it means that **`Lightbulb`** cannot be reused in a context that is free of the concept of **`Button`**.

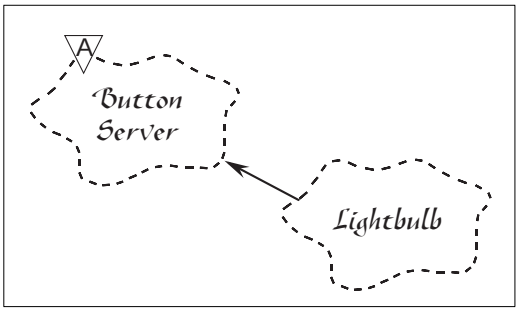

FIGURE 9.
Lightbulb as subclass of ButtonServer

The Adapted Server Pattern

We can correct this situation by employing another one of Gamma's patterns. This one is called ADAPTER. As shown in Figure 10, **`Lightbulb`** can be adapted to **`ButtonServer`** by deriving **`LightbulbButtonServer`** from **`ButtonServer`** and having it contain the **`Lightbulb`**.

1. A class category is a logical grouping of classes which, in Booch's method, is the granule of release, and the granule of reuse.

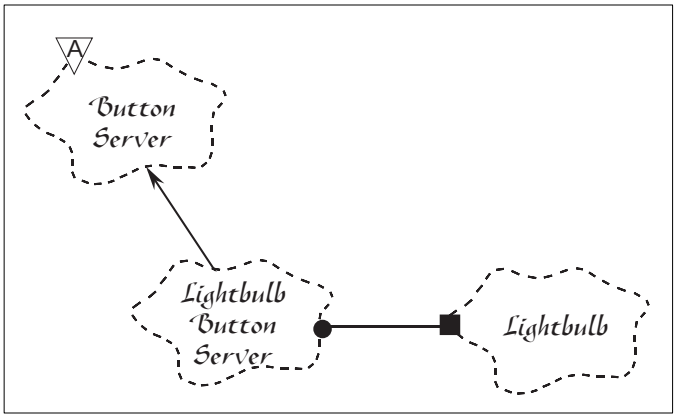

FIGURE 10.
Lightbulb Adapted to ButtonServer.

The conjoined use of ADAPTER and ABSTRACTSERVER is so common, that it is a pattern in its own right. I call it: ADAPTEDSERVER.

This design completely separates **`Button`** from **`Lightbulb`**. Either may be reused without having to drag the other along.

Comparative Analysis of the Solutions

Which of these two solutions is better? The INTELLIGENTCHILDREN approach requires somewhat less code, and fewer classes and objects. There are only two objects, the derivative of **`Button`**, and the **`Lightbulb`**. Whereas the ADAPTEDSERVER requires three objects, the **`Button`**, the **`LightbulbButtonServer`** and the **`Lightbulb`**. Moreover, INTELLIGENTCHILDREN uses fewer CPU cycles than ADAPTEDSERVER because fewer messages need to be passed. So from the point of view of run time efficiency, memory efficiency and development time efficiency, the INTELLIGENTCHILDREN pattern is the winner.

However, ADAPTEDSERVER is a more flexible solution. When this pattern is used, any **`Button`** object can be used to control any derivative of **`ButtonServer`**. Thus, while the INTELLIGENTCHILDREN solution locks the relationship between a particular **`Button`** object and the object it controls, the ADAPTEDSERVER solution allows many different objects to be controlled by the same **`Button`** at different times.

Thus, the choice of pattern to use involves an engineering trade off. Since the speed and complexity overhead of ADAPTEDSERVER is not very high, it will normally be worth the benefit of the extra flexibility. But where memory, CPU and programmer resources are exceedingly tight, it may be best to sacrifice flexibility and use the INTELLIGENTCHILDREN pattern.

Regarding Pragmatism

I think it is important to notice that the most straightforward solution of all was the one represented in Figure 3, the one which I faulted as being most like a procedural design. Pragmatism would seem to dictate that we opt for this solution because it is the simplest. However, simplicity of design does not necessarily relate to simplicity of maintenance, or to simplicity of reuse. Rather, in order to create a design that is maintainable and reusable, some conceptual effort must be applied.

The reason that software applications become unmaintainable and/or unreusable, is that they become cross-linked. There are too many dependencies traversing the design tying one part of the design to another. The net result is the tangled web that so many of us have come to expect from a well evolved software design.

If we are to manage the complexity of these cross-links, then we must provide the infrastructure in the design that will afford that management. The most straightforward design, the design that concentrates only upon functionality, and does not consider maintainability and reusability, will not provide those affordances. Thus, pragmatism must be moderated to some extent.

There is a cost to applying object-oriented design. An good object-oriented design will generally be more complex than a procedural design for the same application. This extra complexity represents the infrastructure needed to manage the dependencies within the software. We should not begrudge this extra complexity. Rather, we should appreciate that it will make the job of maintaining and reusing the software simpler.

Conclusion

This article has shown two methods for separating clients from servers. The INTELLIGENTCHILDREN pattern employs abstract polymorphic interfaces in the client and expects the server to be manipulated by a derivative of the client. The ADAPTEDSERVER pattern places abstract polymorphic interfaces in an abstract class that is contained by the client, and a derivative of which contains the server. In both cases, the key to breaking the dependency between the Client and the Server is in the employment of abstract polymorphic interfaces.