Issue Date: FoxTalk March 1996

Extend and Adapt Your Classes with BRIDGEs

Steven Black
steveb@stevenblack.com

This is the first of a series of articles on object-oriented design patterns and how to apply them in Visual FoxPro. The study of design patterns is an exciting new fad; one that produced some of last year's most insightful computer books, a few of which are listed at the end of this article.

In the first few articles of this series, I'll present examples of terrific object-oriented design patterns that you can probably use immediately. In the months ahead, I'll occasionally pause on the wider issues of pattern usage and pattern coordination.

What is a design pattern?

A *pattern* is a recurring solution to a particular problem. A *design pattern* is a general, but multifaceted, abstraction of the solution and its applicability. Patterns are found wherever several classes and their instances collaborate as a system. You don't need to "know" about patterns to unconsciously create and use them. Some design patterns, when applied under appropriate conditions to solve the correct sort of problem, have known tendencies to be stable, scaleable, and coherent. These patterns are just now being identified, analyzed, and cataloged.

A design pattern defines the parts, collaborations, and responsibilities of classes and instances used in a software subsystem. Thus they abstract the subsystem above the level of classes, instances, and code. Patterns are all about architectures, their component structures, and all their nuances. Abstracting known good solutions into patterns, and cataloging of pattern-based design experience, looks promising as a source of design guidance. If anything, it gives good insight into the balance of forces surrounding a particular problem.

Pattern language gives us a common shared vocabulary about patterns and their implementations. Pattern language seeks to capture and communicate object-oriented design experience. A shared understanding of pattern language can vastly clarify communication among developers in spite of the complexity of the underlying software. So patterns serve as a guide to the sensible design of software building blocks, and they help us better communicate and cope. For more on patterns and specific design patterns, refer to the references at the end of this article.

The BRIDGE pattern

BRIDGE is a scale-independent structural pattern used in many object-oriented situations. The BRIDGE pattern often serves as a player in other design patterns because it defines an *abstract coupling*, meaning how objects interact.

Intent

The BRIDGE decouples an object's public interface ("form") from its implementation (its "function"), using two (or more) objects where otherwise one might less flexibly suffice. Separating an object's programming interface from its implementation means that both form and function objects can be subclassed, extended, or substituted independently. The need for this sort of inherent flexibility is usually why a BRIDGE pattern occurs.

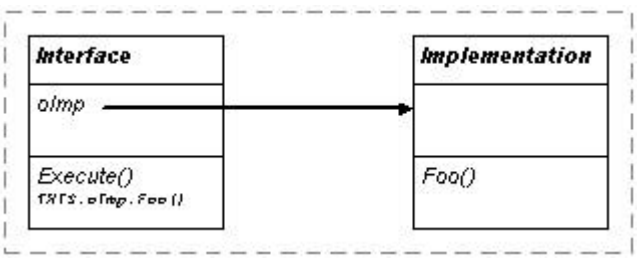
Also known as

HANDLE AND BODY or ENVELOPE AND LETTER are good descriptive names -- the dual-object nature of BRIDGE patterns is well conveyed, as is the tight coupling usually found between the programming interface and implementation objects.

Motivation

A BRIDGE structure is an encapsulated system of players: one (or more) "programming interface" object and one (or more) implementation object. See **Figure 1** for an OMT notation class diagram. The relationship between them, as controlled by the oIMP object reference, is the bridge. This BRIDGE pattern in *Codebook 3.0* hangs on an aptly named member property. Any of the cDataBehavior subclasses may be substituted.

Figure 1. OMT notation class diagram.



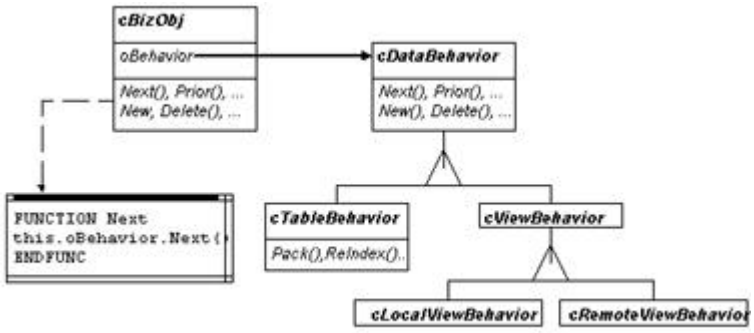
Why use a BRIDGE? A class may have several possible implementations. This seems true of *reusable* classes, especially those that service a variety of clients. The usual way to manage this is by extending the class hierarchy using inheritance. After all, inheritance is usually the first mechanism most folks use to reuse their classes. And why not? In the early going, inheritance is always a big success.

At the limit, however, inheritance leads to sclerosis -- it doesn't scale particularly well. Simple, single-object inheritance binds interface and implementation -- you have only one package to transform into a subclass. This means that interface and implementation aspects can't be independently varied without extending the class hierarchy. But class hierarchies naturally loses their inherent adaptability with size and as the number and variety of clients increases. Why? It's due to side effects, partly. The probability of side-effects in clients increases with the number and variety of those clients, and correspondingly with the current state of the hierarchy. In a giant class hierarchy, making a small change near the top can be very expensive. Occasionally this will limit what can reasonably be done at a given level in a class hierarchy.

The BRIDGE pattern mitigates class sclerosis by decoupling interfaces from implementations, and putting the abstraction and implementation in separate class hierarchies. This allows the interface and implementation classes to vary independently, and the inherent substitutability of subclasses makes the structure intrinsically adaptable (and hence reusable). The bridge in all this is the *relationship* between the interface and the implementation objects that together form a self-supporting system.

The workings among the players in a BRIDGE should be direct and simple: the interface object forwards client requests to the appropriate implementation object. Tight coupling within the BRIDGE structure is both common and desirable. The tight coupling is much easier to manage when all the implementation objects come from the same class, as illustrated in the *Codebook* example in **Figure 2**.

Figure 2. Implementation objects that come from the same class.



Applicability

The BRIDGE pattern is independent of scale and found nearly everywhere within object systems, including within other common software design patterns. Look for BRIDGE when you anticipate future extensions to an object's form or function. In this case, separating form from function can lead to new adaptations without affecting existing client code:

- You want to avoid extending a class hierarchy for the sake of new implementations. The BRIDGE pattern is an effective alternative to an undesirable explosion in the number of subclasses in a given class hierarchy.
- You want to choose a specific behavior at run-time. Using a BRIDGE can drastically lessen the need for design-time divine insight about future applications of this class.
- You need to improve an existing design. The separation of form and function duties is normally invisible to clients, so a BRIDGE object can usually be successfully substituted into existing but inflexible code.
- You want to defer object overhead expenses until they are needed instead of paying in full up-front. A smart BRIDGE can provide this "pay-as-you-go" behavior by instantiating its implementation objects only when (and if) required.
- Hiding implementation details is desirable in spite of the need to fully disclose a programming interface. You can give other developers what they need to integrate your objects without divulging proprietary implementation code.
- Reusing implementation code is made difficult by the need to repeatedly adapt it's programming interface. You can more easily serve diverse clients if the extensibility of an object's programming interface is separate from it's functionality.
- A backpointer from the implementation to the interface object is occasionally desirable. This implies, however, that this implementation object must possess an interface of its own, which is easier to manage if the implementation object is itself a bridge. Bridges, you see, are almost infinitely scaleable and applicable.

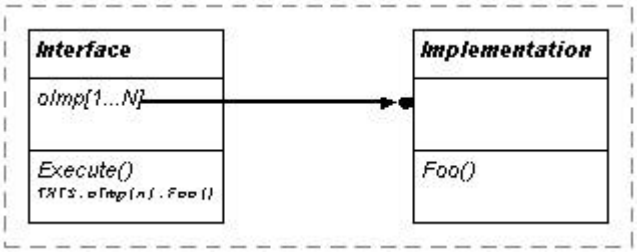
Visual FoxPro samples

Now we'll look at three simple ways BRIDGE patterns can be built in FoxPro: First with member properties. Second we'll do more flexible implementations using member arrays, and third using object composition within containers.

Member Property BRIDGE

One way to build a BRIDGE is diagrammed in **Figure 4**. Here an interface member property contains a reference to an implementation object.

Figure 4. BRIDGE diagram.



In general terms, here's how such systems are constructed in VFP. What follows is a simple illustrative class that provides WAIT WINDOW services.

Listing 1. A simple BRIDGE using a member property to reference the implementation object.

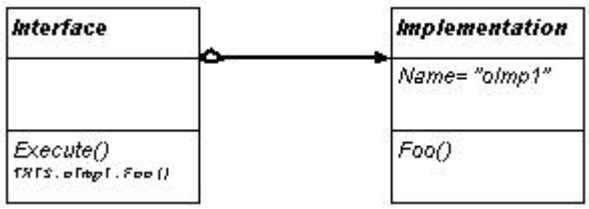
```
XX= CREATEOBJECT( "WaitMsgServer")
XX.Execute("This, for now, is in a WAIT WINDOW")
DEFINE CLASS WaitMsgServer AS MsgInterface
 FUNCTION Init( txPassed)
  *-- Load an interface object
  THIS.aImp= CREATEOBJECT("WaitMsg")
 ENDFUNC
 FUNCTION Execute( txPassed)
  *-- Pass the request along
  THIS.aImp.Execute( txPassed)
 ENDFUNC
ENDDEFINE
DEFINE CLASS WaitMsg AS MsgImplementation
 FUNCTION Show( txPara1)
  WAIT WINDOW THIS.cMessage
 ENDFUNC
ENDDEFINE
DEFINE CLASS MsgInterface AS CUSTOM
 *-- Abstract message interface class
 aImp= .NULL.
 FUNCTION Execute( txPassed)
   *-- Abstract
 ENDFUNC
ENDDEFINE
DEFINE Class MsgImplementation AS Custom
 *-- Abstract message implementation class
 cMessage= ''
 FUNCTION Execute( tcPassed)
  THIS.cMessage=tcPassed
  THIS.Show()
 ENDFUNC
 FUNCTION Show
   *-- Abstract
 ENDFUNC
ENDDEFINE
```

Another example of this sort of BRIDGE is found in Codebook where the cBizobj (business object) class uses an object of the cDataBehavior class to implement the usual table navigation and record-processing functions. The diagram in **Figure 2** illustrates this.

Member Array BRIDGE

Starting from the Member Property BRIDGE, it's a short stretch to provide multiple simultaneous implementations by using multiple member properties or, as described below, member arrays. The diagram in **Figure 3** shows the use of multiple arrays that map to multiple implementations.

Figure 3. Diagram showing use of multiple arrays.



Listing 1 is modified in Listing 2 to support four different types of dialog boxes to extend the legacy WAIT WINDOW capability:

Listing 2. Many implementations can be connected to an interface array member property.

```
DEFINE CLASS WaitMsgServer AS MsgInterface
 FUNCTION Init
  *-- Load interface objects
  THIS.aImp[1]= CREATEOBJECT("WaitMsg")
  THIS.aImp[2]= CREATEOBJECT("RegularMsg")
  THIS.aImp[3]= CREATEOBJECT("InfoMsg")
  THIS.aImp[4]= CREATEOBJECT("WarningMsg")
  THIS.aImp[5]= CREATEOBJECT("ErrorMsg")
  *-- Supporting the legacy singular WaitMsg capability
  THIS.oImp= aImp[1]
 ENDFUNC
 FUNCTION Execute( txPassed, tnMessageType)
  THIS.aImp[tnMessageType].Execute( txPassed)
 ENDFUNC
ENDDEFINE
DEFINE CLASS WaitMsg AS MsgImplementation
 FUNCTION SHOW( txPara1)
  WAIT WINDOW THIS.cMessage
 ENDFUNC
ENDDEFINE
DEFINE CLASS RegularMsg AS MsgBoxImplementation
 cTitle= "My Application"
ENDDEFINE
DEFINE CLASS InfoMsg AS RegularMsg
 nIcon= 64
ENDDEFINE
DEFINE CLASS WarningMsg AS InfoMsg
 nIcon= 48
ENDDEFINE
DEFINE CLASS ErrorMsg AS WarningMsg
 nIcon= 16
 nButtons= 5
ENDDEFINE
DEFINE CLASS MsgInterface AS CUSTOM
 *-- Abstract message interface class
 DIMENSION aImp[4]
 aImp[1]= .NULL.
 aImp[2]= .NULL.
 aImp[3]= .NULL.
 aImp[4]= .NULL.
 *-- Virtual
 FUNCTION Execute( txPassed)
ENDDEFINE
DEFINE CLASS MsgBoxImplementation AS MsgImplementation
 nIcon= 0
 nButtons= 0
 FUNCTION Show
  =MessageBox( THIS.ctext, THIS.nIcon+THIS.nButtons, THIS.cTitle)
 ENDFUNC
ENDDEFINE
DEFINE Class MsgImplementation AS Custom
 *-- Abstract message implementation class
 cMessage= ''
 FUNCTION Execute( tcPassed)
  THIS.cMessage=tcPassed
  THIS.show()
 ENDFUNC
 *-- Virtual
 FUNCTION Show
 ENDFUNC
ENDDEFINE
```

Containership BRIDGE

Containership implementations are similar to array member implementations. In Visual FoxPro, containership is superbly implemented, so containership BRIDGEs are easy to create and manage. Useful are the Controls() array, the PARENT keyword, SetAll(), and AMEMBERS(,,2), which make it possible to manage containership nesting.

The containership relationship is illustrated in Figure 4, and Listing 3 is an illustrative code example.

Listing 3. Multiple implementations instantiated in an interface container.

```
DEFINE CLASS MsgServer AS MsgInterface
 FUNCTION Init
 *-- Load an interface object with implementations.
 *--   Exercise for the reader: Imagine delaying
 *--   the AddObject calls until actually needed.
 THIS.AddObject( "msgWaitWindow","WaitMsg")
 THIS.AddObject( "msgRegular",   "RegularMsg")
 THIS.AddObject( "msgInfo",      "InfoMsg")
 THIS.AddObject( "msgWarning",   "WarningMsg")
 THIS.AddObject( "msgError",     "ErrorMsg")
 ENDFUNC
 FUNCTION Execute( tnPassed, tcMessage)
  *? I don't recommend doing it quite like this --
  *? This simple example assumes you know the number
  *? of the implementation object.  In a perfect but
  *? (less concise) example, .Execute(x,y) could accept
  *? an x of type "C", as in
  *? .Execute("Warning", )
  *?
  THIS.Controls(tnPassed).Execute(tcMessage)
ENDFUNC
ENDDEFINE
DEFINE CLASS WaitMsg AS MsgImplementation
 FUNCTION SHOW( txPara1)
  WAIT WINDOW THIS.cMessage
 ENDFUNC
ENDDEFINE
DEFINE CLASS RegularMsg AS MsgBoxImplementation
 cTitle= "My Application"
ENDDEFINE
DEFINE CLASS InfoMsg AS RegularMsg
 nIcon= 64
ENDDEFINE
DEFINE CLASS WarningMsg AS InfoMsg
 nIcon= 48
ENDDEFINE
DEFINE CLASS ErrorMsg AS WarningMsg
 nIcon= 16
 nButtons= 5
ENDDEFINE
DEFINE CLASS MsgInterface AS Container
 *-- Abstract message interface class
 *?  Hardcoded dimension
 DIMENSION aImp[4]
 aImp[1]= .NULL.
 aImp[2]= .NULL.
 aImp[3]= .NULL.
 aImp[4]= .NULL.

 FUNCTION Execute( txPassed)
  *-- Abstract method
  RETURN 0
ENDDEFINE
DEFINE Class MsgImplementation AS Custom
 *-- Abstract message implementation class
 cMessage= ''
 FUNCTION Execute( tcPassed)
  THIS.cMessage=tcPassed
  THIS.Show()
 FUNCTION Show
  *-- Abstract method
  RETURN 0
ENDDEFINE
DEFINE CLASS MsgBoxImplementation AS MsgImplementation
 nIcon= 0
 nButtons= 0
 FUNCTION Show
  =MessageBox( THIS.ctext, THIS.nIcon+THIS.nButtons, THIS.cTitle)
 ENDFUNC
ENDDEFINE
```

What to expect from BRIDGE patterns

On balance, classes built as BRIDGEs are easier to extend and adapt since their interfaces and implementations can be subclassed independently. Moreover, the BRIDGE pattern is easy to grasp and is easily communicated among developers and designers.

Even though programming interfaces and implementations are separate, a high degree of *cohesion* (coupling) between participants is to be expected unless steps are taken to abstract their relationship. The BRIDGE pattern scales nicely, and it's not unusual to find a BRIDGE containing other BRIDGEs. This is a good rule of thumb: when creating a new class, always consider making it a BRIDGE first.

Next month we'll talk further about abstracting the relationship between objects when we discuss OBSERVER and MEDIATOR patterns.

Sidebar: References

Black, S. (1995) *Pattern Implementation in VFP*, European Visual FoxPro Developers Conference '95 session notes, E-PATT, Frankfurt, Germany. And also: GLGDW '95 session notes, #30, Milwaukee, WI.

Coplien, J, and Schmidt, D (1995) *Pattern Languages of Program Design*. Reading, MA. Addison Wesley. ISBN 0-201-60734-4.

Gamma, E., Helm, R., Johnson, R, and Vlissides, J. (1994) *Design Patterns, Elements of Object Oriented Software*. Reading, MA. Addison Wesley. ISBN 0-201-63361-2

Pree, W (1995) *Design Patterns for Object Oriented Development*. Reading, MA. Addison Wesley and ACM press. ISBN 0-201-42294-8.