This paper was originally presented at the Southwest Fox conference in Gilbert, Arizona in October, 2010. http://www.swfox.net

Design Patterns in Visual FoxPro

Rick Borup
Information Technology Associates
701 Devonshire Dr, Suite 127
Champaign, IL 61820
Voice: (217) 359-0918
Fax: (217) 398-0413
Email: rborup@ita-software.com

Design patterns are abstracted solutions that can be applied to solve common programming challenges in object-oriented software systems. Although it's been fifteen years since the seminal book “Design Patterns: Elements of Reusable Object-Oriented Software” was published, many developers are probably still not taking advantage of all that design patterns have to offer. While design patterns may most often be thought of in the context of languages like Java or C#, they are equally applicable and useful in Visual FoxPro. This session explores several common design patterns and shows you how to apply them to your work in Visual FoxPro.

Table of Contents

What is a design pattern?

A design pattern is a generalized approach to solving a recurring problem. The word problem here is used in the sense of a challenge or a requirement. The point is that design patterns provide a way of addressing commonly recurring situations in ways that are known to work and which carry known benefits and trade-offs.

Design patterns can be found in all sorts of disciplines including architecture, engineering, construction, and urban planning just to name a few. In the context of computer software design there are design patterns for application architecture, database design, and of course object-oriented programming (OOP), which is the domain we're interested in here.

Object-oriented software designs are built on a foundation of classes. The job of an OOP software designer is to design the appropriate classes and put them together in such a way as to implement the desired behavior of the software, while at the same time imposing minimum constraints on the inevitable future changes to the system. The guiding principle is to design for reusability and extensibility.

Object-oriented software design patterns describe the relationship, collaboration, and responsibilities among a group of classes. Design patterns are therefore useful to the OOP software developer because they provide a structure for how groups of classes can be made to interact in desired ways.

Why learn design patterns?

There are at least three benefits to learning about design patterns even if you are already an experienced OOP software developer.

The first is that you will be able to recognize design patterns when you see them in existing software. Whether implemented elegantly or poorly, you'll begin to see where patterns have been used and perhaps where they could have been used more effectively. Keep in mind that the existing software you're looking at might be your own! Learning design patterns will help you see opportunities to refactor your own code into a more effective and extensible design.

This is particularly true for Visual FoxPro developers. Because of VFP's procedural roots, VFP programs can and often do contain a mixture of procedural and OOP design elements. I know this is true of my own work and I suspect it's true for many other VFP developers as well. On the other hand, design patterns by their very nature enforce a pure OOP approach to implementing a solution. Once you've learned design patterns you may look at your older work in an entirely new (and perhaps not so flattering) light.

A second benefit is that you will learn to recognize opportunities to use design patterns when you are developing new software. The ability to see potential uses for design patterns improves the way you approach the design process as a whole. The result is a better, more flexible software design from the ground up.

Finally, you'll acquire a common vocabulary for discussing OOP designs with other developers. This can be especially important if you work in a team environment because the lack of a common frame of reference makes communication much more difficult. To paraphrase an example given in the book Design Patterns Explained (see References), imagine the difficulty one carpenter would have explaining to another carpenter how to join two pieces of wood together in a certain way if he had to describe each individual angle and cut rather than being able to simply refer to it as a dovetail joint. The name "dovetail joint" describes a design pattern in carpentry whose structure is instantly familiar to both carpenters. The same goes for OOP design patterns: giving names to patterns that have a known structure and that solve know problems makes it much easier to discuss them with other developers.

Classifying design patterns

The seminal book on OOP design patterns is Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides, commonly referred to as the Gang of Four or GoF for short. This book catalogs 23 design patterns, which are classified into one of three categories according to their purpose.

- Structural design patterns demonstrate ways in which classes can be combined to meet various requirements.
- Creational design patterns deal with ways of creating objects.
- Behavioral design patters are concerned with the ways in which a group of objects interact with one another.

Design pattern notation

In order to talk about design patterns you need to become familiar with the notation commonly used to describe them.

The GoF book, which was first published in 1995, uses Object Modeling Technique (OMT) to illustrate design patterns. More recent books on the subject tend to use the Unified Modeling Language (UML), which is similar but different enough to merit separate discussion. Because the GoF book is still the single most authoritative reference for OOP design patterns, I have chosen to use OMT notation to illustrate the structure of the design patterns discussed in this paper.

Elements of OMT notation

The fundamental element of any OOP design pattern is the class. In an OMT class diagram, a class is represented by a rectangle. The name of the class is listed in bold at the top of the rectangle. Abstract class names are shown in italic font while concrete class names are in shown in normal font.

If any class methods are relevant to the diagram, a line is drawn below the class name and the method names are then listed below the line. Instance variables (properties in Visual FoxPro) can also be shown, separated by another line. Finally, if they're relevant to the diagram, the return type of a method and the type of a variable can precede its name.

Figure 1: In OMT, a class is illustrated by a rectangle showing its name along with important methods and variables.

Design patters are all about the relationships among the classes in the design. OMT class diagrams use different line notations to illustration the different types of relationships.

- Inheritance is illustrated by a sold line with a triangle pointing from the subclass to its parent class. Inheritance represents an "is a" relationship, with increasing specialization as you move down the class hierarchy.
- Acquaintance is illustrated by a solid line with an arrow head pointing from the class that maintains the reference to the class that is referenced. Acquaintance represents a "knows a" or "uses a" relationship between classes.
- Aggregation is illustrated in the same way as acquaintance except the line has an open (un-shaded) diamond shape at its base. Aggregation represents a "has a" relationship. It can be thought of as a stronger form of acquaintance where the owning object determines the lifetime of the owned object.
- Instantiation is illustrated by a dashed line with an arrow head pointing from the class that does the instantiation to the class that is instantiated.[1] Instantiation represents a "creates a" relationship.
- Pseudo code is illustrated by a dashed line originating with an open circle next to the method to which it pertains, connected to a rectangle with a folded corner containing the pseudo code. [2]

Figure 2 illustrates how these relationships appear in an OMT class diagram.

[1] This convention for illustrating instantiation is an extension to OMT adopted by the GoF book.

[2] This is also a GoF extension to OMT.

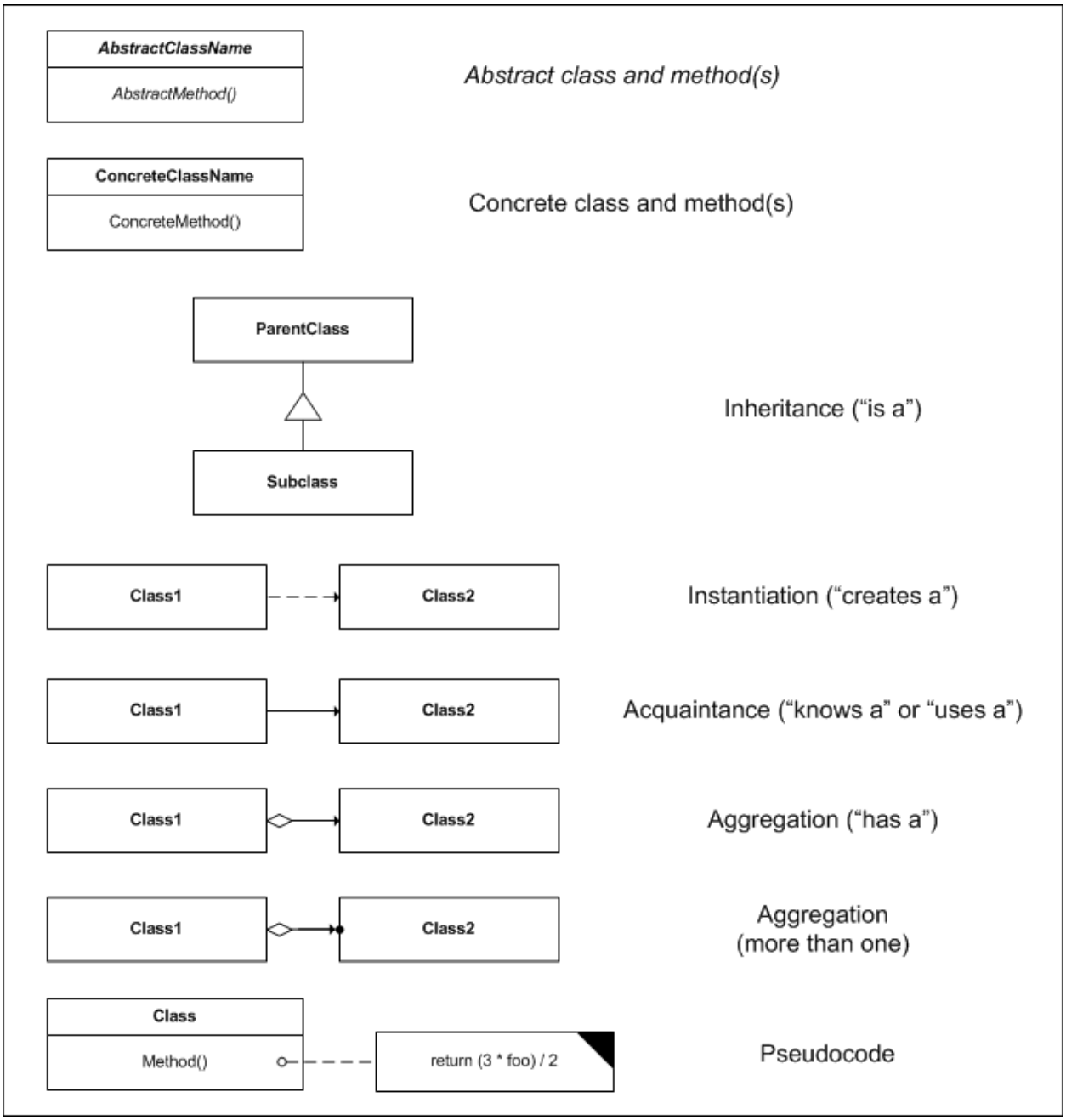

Figure 2: OMT uses different types of lines to illustrate different relationships among classes.

How to read an OMT class diagram

As an exercise, consider the class diagram in Figure 3. How would you describe in words what this diagram represents?

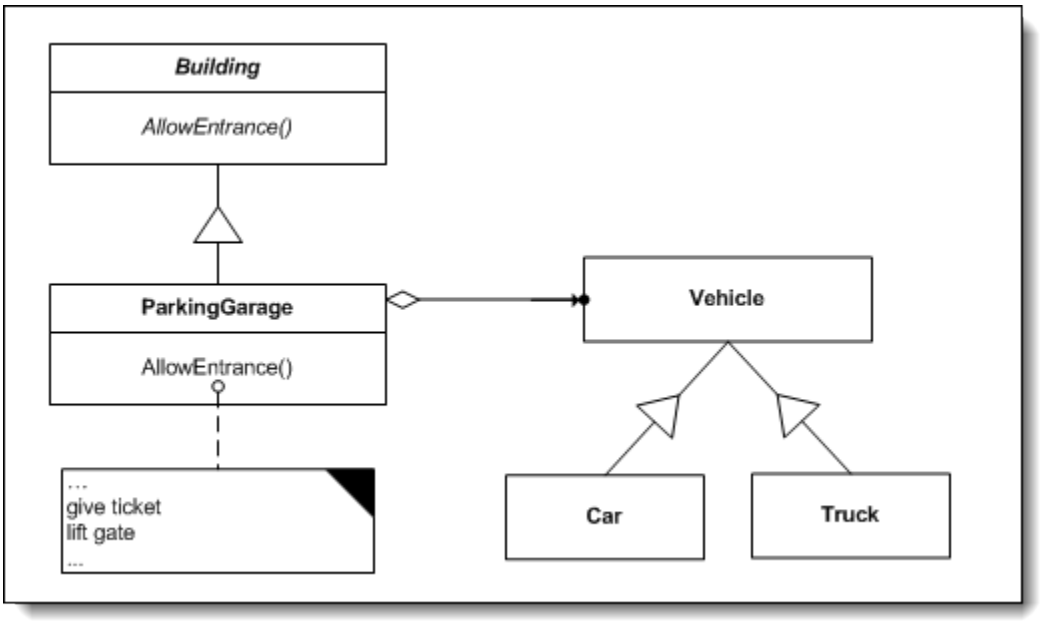

Figure 3: An OMT class diagram is a visual representation of one or more classes and the relationship between them.

This class diagram shows a concrete class named ParkingGarage, which is subclassed from the abstract Building class. The ParkingGarage class implements the AllowEntrance() method by, among other things, giving a ticket and lifting a gate. The ParkingGarage class also "has" (maintains a reference to) one or more vehicles, which, via inheritance, can be either cars or trucks.

It may take a little effort to learn how to read and understand class diagrams, but once you become familiar with them you'll get a lot more value out of any book or article about design patterns. If you encounter a diagram that uses UML instead of OMT, remember that the concepts are the same, it's only the notation that is different.

The Strategy pattern

The Strategy pattern is the one I find myself using most often in my own work. Once you become familiar with it you may find the same is true for you. In any case, it's a good pattern to start with.

Here's the definition of the Strategy pattern.[3]

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

[3] All the design pattern definitions in this paper are quoted from the GoF book, which is cited in the References section of this paper.

This sounds like pretty heady stuff, but what does it really mean?

An algorithm is a method for solving a problem. A family of related algorithms is therefore a set of methods for solving the same or similar problems in different ways. Encapsulate each one means to place each algorithm in its own class, and make them interchangeable means make each one use the same interface. In other words, although they're separate, you can interact with any one of the algorithms in the same way as with any of the others.

Figure 4 is the class diagram for the Strategy pattern. Note that the Context class maintains a reference to the Strategy class; in other words, the Context "has a" Strategy. Notice also that the various solution algorithms are implemented as subclasses that inherit from the abstract Strategy class. The Strategy pattern is classified as a behavioral pattern because it deals with the way in which its classes interact with one another.

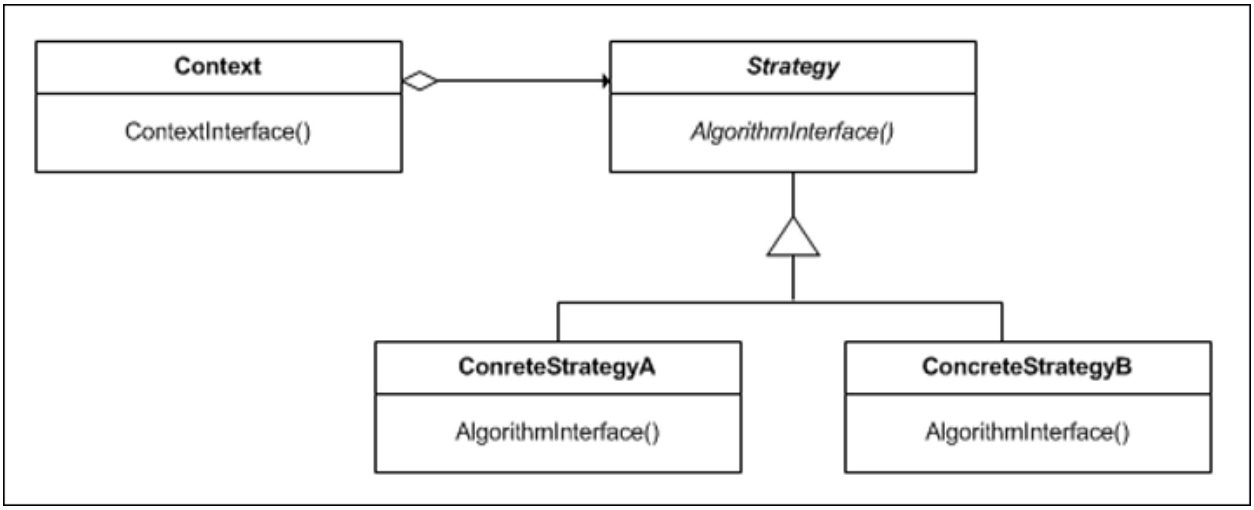

Figure 4: The structure of the Strategy pattern.

Consider an example. I used to work in the commercial banking industry, where a frequent requirement was to write code to calculate the amount of interest due on a deposit account. Interest is commonly calculated either as simple interest or as compound interest. Simple interest is a straight-line calculation in which a periodic rate is multiplied by the number of periods and the result is multiplied by the account balance. Compound interest involves calculating "interest on interest" and requires a loop structure that iterates over a the appropriate number of periods, incrementing the result as it goes along.

This scenario is perfectly suited for a Strategy pattern because, although the two methods of calculating the interest differ from one another, each requires the same data (a balance, a rate, and a period of time) and each returns the same type of value (a numeric amount). In other words, the two algorithms share the same interface.

Deriving the Strategy pattern from procedural code

Because this is the first design pattern presented, and because this is a paper intended for Visual FoxPro developers, many of whom have been working with FoxPro since the procedural days before it became an OOP language in VFP 3.0, I want to demonstrate the Strategy pattern by showing how it might evolve from purely procedural code of the kind we used to write in FoxPro for DOS (and for that matter can still write in VFP). To do this, I present several examples where each one evolves from the previous one, with the last one being an implementation of the actual Strategy pattern. I hope this approach helps you gain a solid understanding of what the Strategy design pattern is and how it works.

Listing 1 illustrates a simplistic solution to calculating interest in a purely procedural manner. It is intended to represent something you might find in an old FoxPro for DOS (FPD) program where there were no such things as classes.

Listing 1: A procedural solution to calculate simple or compound interest. The rate is divided by 100 so it can be passed in as a percentage rather than a decimal value.
(Session code: \Strategy\CalculateInterest_Procedural1.prg)

```
FUNCTION CalculateInterest
PARAMETERS tnBalance, tnRate, tnPeriod, tcType
lnInterest = 0.00
IF tcType = "simple"
  lnInterest = tnBalance * ( ( tnRate / 100) / 365) * tnPeriod
ELSE
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 365)
  ENDFOR
ENDIF
RETURN lnInterest
```

The code in Listing 1 is quite straight forward but suffers from some limitations that don't become apparent until the requirements begin to grow more complex. For example, the only kind of compound interest the code in Listing 1 can calculate is daily compound interest, in which the annual rate is divided by 365 to get the periodic rate (the daily rate) and the period is expressed in days. What happens when the bank decides to offer an account that pays quarterly compound interest or monthly compound interest? The periodic rate is then based on a quarter or a month, and the period is no longer expressed in days. The entire function needs to be revised to accommodate these additional requirements. One solution to this revision is shown in Listing 2.

Listing 2: The function can now calculate interest in four different ways.
(Session code: \Strategy\CalculateInterest_Procedural2.prg)

```
FUNCTION CalculateInterest
PARAMETERS tnBalance, tnRate, tnPeriod, tcType
lnInterest = 0.00
DO CASE
  CASE tcType = "quarterly"
```

```
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 4)
    ENDFOR
  CASE tcType = "monthly"
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 12)
    ENDFOR
  CASE tcType = "daily"
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 365)
    ENDFOR
  OTHERWISE  && simple
    lnInterest = tnBalance * ( ( tnRate / 100) / 365) * tnPeriod
ENDCASE
RETURN lnInterest
```

(It's apparent that we could collapse the three compound interest methods into one and derive the divisor from the type, but I'll leave them separate for the sake of illustration.)

This code works fine, but it's still suffers from the same limitations as the first example. The primary issue is that the code is monolithic. Although in this example the algorithms are at most three lines of code apiece, imagine a more complicated set of algorithms that require many more lines of code for each solution. You can see how the entire function could become fairly long fairly quickly, making it more difficult to maintain. Moreover, if a new dimension is introduced, such as the ability to use a daily rate based either on 365/360 or 365/365, then a single monolithic function becomes even more cumbersome.

The way this code is written, you have to edit the entire function's source code in order to change any of these algorithms or add new ones. This introduces the risk of breaking something that previously was working. Finally, to put it into OOP terminology, both the interface and the implementation are bound up in a single piece of code, which limits maintainability and reusability.

Let's assume you encounter the code in Listing 2 in an old (but working) FoxPro for DOS program and need to implement it in a new Visual FoxPro version of the same application. Your first inclination might be to simply make it a method and wrap it in a class.

Listing 3: This is the same code as in Listing 2, but now wrapped up as a method in a class.
(Session code: \Strategy\CalculateInterest_ObjProc.prg)

```
DEFINE CLASS clsCalculateInterest as Custom
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
lnInterest = 0.00
DO CASE
  CASE tcType = "quarterly"
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 4)
    ENDFOR
  CASE tcType = "monthly"
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 12)
```

```
    ENDFOR
  CASE tcType = "daily"
    FOR lni = 1 TO tnPeriod
      lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 365)
    ENDFOR
  OTHERWISE  && simple
    lnInterest = tnBalance * ( ( tnRate / 100) / 365) * tnPeriod
ENDCASE
RETURN lnInterest
ENDDEFINE
```

Simply wrapping a chunk of procedural code inside a method of a class does not, however, constitute object-oriented design. It merely results in what I call “objectified procedural” code, or pseudo-OOP. The code in Listing 3 still suffers from the same limitations as the previous example. Even though it uses a class, this solution does really not move us any closer to implementing a real Strategy pattern.

The next step along the way toward a true implementation of the Strategy pattern might be to break the different calculations into separate functions. We might also decide to use properties to avoid having to pass so many parameters around.

Listing 4: A first attempt at refactoring the solution into something more like a Strategy pattern.
(Session code: \Strategy\CalculateInterest_OOP.prg)

```
DEFINE CLASS clsCalculateInterest as Custom
nBalance = 0.00
nRate = 0.00
nPeriod = 0
cType = ""
nInterest = 0.00

FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
  WITH this
    .nBalance = tnBalance && Variable type checking omitted
    .nRate = tnRate
    .nPeriod = tnPeriod
    .cType = tcType
    DO CASE
      CASE tcType = "quarterly"
        .CalculateQuarterlyInterest()
      CASE tcType = "monthly"
        .CalculateMonthlyInterest()
      CASE tcType = "daily"
        .CalculateDailyInterest()
      OTHERWISE   && simple
        .CalculateSimpleInterest()
    ENDCASE
 ENDWITH
 RETURN this.nInterest
ENDFUNC

FUNCTION CalculateSimpleInterest()
  WITH this
```

```
    .nInterest = .nBalance * ( ( .nRate / 100) / 365) * .nPeriod
  ENDWITH
ENDFUNC

FUNCTION CalculateQuarterlyInterest()
  WITH this
    .nInterest = 0.00
    FOR lni = 1 TO .nPeriod
      .nInterest = .nInterest + ( .nBalance + .nInterest) * ( ( .nRate / 100) / 4)
    ENDFOR
  ENDWITH
ENDFUNC

FUNCTION CalculateMonthlyInterest()
  WITH this
    nInterest = 0.00
    FOR lni = 1 TO .nPeriod
      .nInterest = .nInterest + ( .nBalance + .nInterest) * ( ( .nRate / 100) / 12)
    ENDFOR
  ENDWITH
ENDFUNC

FUNCTION CalculateDailyInterest()
  WITH this
    .nInterest = 0.00
    FOR lni = 1 TO .nPeriod
      .nInterest = .nInterest + ( .nBalance + .nInterest) * ( ( .nRate / 100) / 365)
    ENDFOR
  ENDWITH
ENDFUNC
ENDDEFINE
```

While this might be considered a step in the right direction, it's still monolithic (a single class) and therefore not yet a true implementation of the Strategy pattern.

Remember when you were a little kid riding in the car on a long trip and you'd bug your parents with the question "Are we there yet?" Or maybe you have your own kids now and you're the one being asked that same question! Anyway, the answer here is, "Yes, we're almost there."

Implementing the Strategy pattern

Review the illustration of the Strategy pattern in Figure 4. Note that, like all design patterns, it consists of two or more separate classes with a defined relationship between them. To implement a Strategy pattern solution to the Calculate Interest problem we need to design multiple classes and organize them in the proper fashion.

As you can see from Figure 4, the Strategy pattern uses inheritance to encapsulate the different algorithms in separate classes that all respond to the same interface. This group of classes is what implements the Strategy. It consists of one abstract class, which defines the

interface, along with two or more subclasses, each of which implements one of the solution algorithms.

The Strategy pattern also involves a Context object that maintains a reference to the Strategy object. At runtime, it's the Context object that holds the values for the algorithm's data as well as determining which specific solution is required. In Visual FoxPro the Context object is likely to be a form, but it could be any object.

The code in Listing 5 shows one way to implement the Calculate Interest solution using a true Strategy pattern. Ordinarily we would not define a Form class in code, but we don't need to show all the overhead of a full form for the sake of an example. Imagine that the form we're talking about here is part of a financial calculator application that a bank employee might run on their desktop computer to help a customer decide which type of account to open.

Listing 5: The interest calculation problem solved with a true Strategy pattern.
(Session code: \Strategy\CalculateInterest_Strategy.prg)

```
*--------------------------------------*
*       "Context" Class                *
*--------------------------------------*
DEFINE CLASS clsFinancialCalculator as Form
nBalance = 0.00
nRate = 0.00
nPeriod = 0
cType = ""

FUNCTION GetInterestAmount()
loStrategy = NEWOBJECT( "cls" + this.cType)
RETURN loStrategy.CalculateInterest( this.nBalance, this.nRate, this.nPeriod)
ENDDEFINE

*--------------------------------------*
*       "Strategy" Classes             *
*--------------------------------------*

*  Interface class (abstract)
DEFINE CLASS clsCalculateInterest as Custom
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod)
  *  Virtual
ENDFUNC
ENDDEFINE

*  Implementation classes ("ConcreteStrategy")
DEFINE CLASS clsSimple as clsCalculateInterest
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod)
  lnInterest = tnBalance * ( ( tnRate / 100) / 365) * tnPeriod
  RETURN lnInterest
ENDFUNC
ENDDEFINE
```



```
DEFINE CLASS clsQuarterly as clsCalculateInterest
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod)
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 4)
  ENDFOR
  RETURN lnInterest
ENDFUNC
ENDDEFINE

DEFINE CLASS clsMonthly as clsCalculateInterest
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod)
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 12)
  ENDFOR
  RETURN lnInterest
ENDFUNC
ENDDEFINE

DEFINE CLASS clsDaily as clsCalculateInterest
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod)
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 365)
  ENDFOR
  RETURN lnInterest
ENDFUNC
ENDDEFINE
```

There are several ways for the Context to determine which class to instantiate in order to provide the correct solution. These include maintaining an array or a collection or even using a CASE statement. In this example, however, I'm using a simple yet elegant approach I first came across in one of Andy Kramek's excellent papers on design patterns (see References), which takes advantage of VFP's ability to compose the name of the desired class on the fly at runtime.

Unseen in this code is the Client, which is the entity that needs to invoke the strategy. If the Context is a Visual FoxPro form, as shown in Listing 5, the Client could be a command button that invokes the form's GetInterestAmount method.

Reviewing the Strategy pattern

At this point you may reasonably be asking, "Was it worth it?" After all, what started out as a simple CASE statement within a single Function has morphed into a solution involving two classes and four subclasses. Did we really gain any tangible benefits by trading the apparent simplicity of the procedural solution for the apparent complexity of the Strategy design pattern solution?

I believe the answer is yes, and I hope you do too.

Let's review: The procedural solution was monolithic and would grow increasingly difficult to maintain as the required number of different solutions grew. In the procedural code the interface and the implementation were all wrapped up in a single block of code, making reusability difficult. In the design pattern solution, the Strategy class and its subclasses are separate implementations that can be individually maintained. The Strategy group of classes can be stored in a separate class library and used by whatever Context object needs it, in this or in any other application. The Context class and the Strategy classes might even be written and maintained by a different developers, because the developer of the form (the Context) does not need to know anything about how the various algorithms (the Strategies) are implemented.

The Bridge pattern

Look again at the class diagram for the Strategy pattern in Figure 4. Mentally factor out the subclasses. What's left is a class that provides an abstraction of a solution that's implemented in another class. This separation of a solution's abstraction from its implementation is a design pattern in its own right. Its name is the Bridge pattern.

Here's the definition of the Bridge pattern.

> Decouple an abstraction from its implementation so that the two can vary independently.

The basic class diagram for the Bridge pattern is shown in Figure 5. Although shown that way in the class diagram, it's not strictly necessary for the Abstraction object (the interface) to maintain a "has a" relationship with the Implementation object. It's sufficient just to have a "uses a" relationship, meaning the lifetime (duration of existence) of the Implementation object can be separate from the lifetime of the Abstraction object.

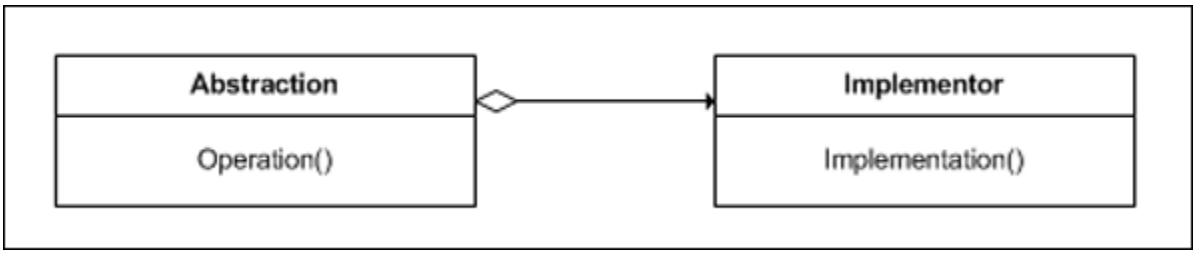

Figure 5: The structure of the Bridge pattern.

Note that the word "abstraction" in this pattern does not refer to an abstract class; it simply refers to the fact that the Operation method, which exists in one class, calls an Implementation method that exists in another class. The Bridge is the relationship between these two classes.

As you can see by comparing Figure 4 to Figure 5, the Strategy pattern incorporates the Bridge pattern. The concept of using design patterns in combination with one another is

fundamental to object oriented software design. The Bridge pattern is one of the most basic structural patterns, and you'll likely recognize it in many places.

As an example of the Bridge pattern, consider the financial calculator form from the previous example. In the United States, banks are required to disclose the Annual Percentage Yield (APY) in addition to the interest rate. This requirement was put in place to help customers compare competing products from different financial institutions in order to find the best return on their money. Comparing interest rates alone can be misleading because a lower nominal rate, compounded daily, might actually yield more interest than a higher nominal rate calculated as simple interest. The Annual Percentage Yield calculation evens out these differences by expressing the return on an investment as a normalized rate based on a period of one year.

The formula for calculating the APY is:

APY = (1 + (i / n))^n – 1

where i is the nominal interest rate (i.e., the advertised rate) and n is the period. Since the APY is always based on one year, n is 1 for simple interest, 4 for quarterly compound interest, 12 for monthly compound interest, or 365 for daily compound interest.

We can implement this calculation in Visual FoxPro like this

```
lnAPY = ( 1 + ( lnRate / lnPeriod ) )**lnPeriod – 1
```

In this code the rate has to be passed in as a decimal, for example .05 for 5% interest (Hah! Those were the days, right?) and the APY is returned as a decimal, for example .0513.

If we want to be able to pass the rate as a percentage and get a percentage back in return we simply have the modify the calculation accordingly. Listing 6 shows this code as a method on a class.

Listing 6: The APY calculation.

```
DEFINE CLASS clsCalculateAPY as custom
FUNCTION CalculateAPY( lnRate, lnPeriod)
lnAPY = ( 1 + ( ( lnRate / 100) / lnPeriod ) )**lnPeriod - 1
RETURN ROUND( lnAPY * 100, 2)
ENDDEFINE
```

So, what about the Bridge we were supposed to be building?

Again, think back to the financial calculator form from the Strategy pattern example. In addition to being able to calculate the dollar amount of interest on a given balance using the Strategy pattern, we now also want the bank employee to be able to quote the APY. We can implement this as a Bridge pattern so that the form (the Abstraction) does not need to

know anything about the APY formula (the Implementation). The code to accomplish this is shown in Listing 7.

Listing 7: The APY calculation can be implemented using a Bridge pattern.
(Session code: \Bridge\clsCalculateAPY.prg)

```
*--------------------------------------*
*       "Abstraction" Class            *
*--------------------------------------*
DEFINE CLASS clsFinancialCalculator as Form
nBalance = 0.00
nRate = 0.00
nPeriod = 0
cType = ""
FUNCTION GetAPY()
loBridge = NEWOBJECT( “clsCalculateAPY”)
RETURN loBridge.CalculateAPY( this.nRate, this.nPeriod)
ENDDEFINE

*--------------------------------------*
*       "Implementation" Class         *
*--------------------------------------*
DEFINE CLASS clsCalculateAPY as Custom
FUNCTION CalculateAPY( lnRate, lnPeriod)
lnAPY = ( 1 + ( ( lnRate / 100) / lnPeriod ) )**lnPeriod - 1
RETURN ROUND( lnAPY * 100, 2)
ENDDEFINE
```

Remember that the definition of the Bridge pattern includes the phrase “so that the two [the Abstraction and the Implementation] can vary independently”. In this example, the form can be subclassed and a call to its GetAPY() method still returns the desired result via the Bridge that exists in the parent form class. In the same way, if the calculation of the APY were to change, class clsCalculateAPY could be modified or replaced and the calculation would still work without any change to the form.

The Chain of Responsibility pattern

The Chain of Responsibility pattern is another behavioral pattern for which you may find frequent use. Like the Strategy pattern, its purpose is to provide a way in which any one of two or more different objects can handle a request.

Here's the definition of the Chain of Responsibility pattern.

> Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

While the intent of the Chain of Responsibility pattern is similar to the intent of the Strategy pattern, they differ completely in their structure and implementation. Figure 6 illustrates the class diagram for the Chain of Responsibility pattern.

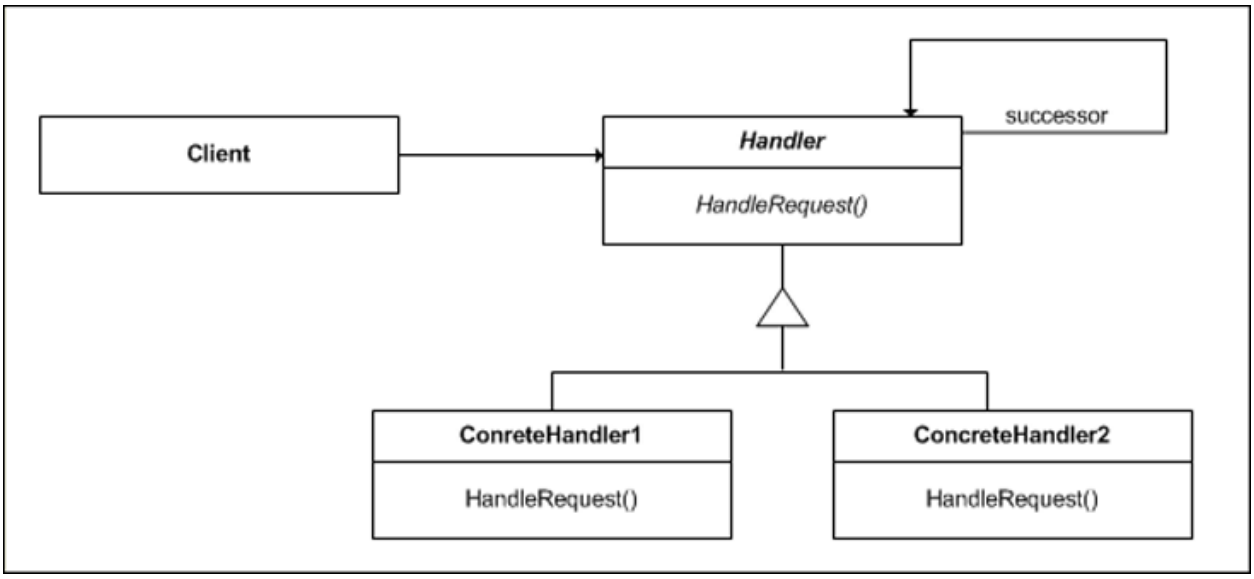

Figure 6: Class diagram of the Chain of Responsibility pattern.

Although the class diagram shows it that way, in actual practice the objects that can handle the request do not need to be subclassed from the same abstract class. In fact, the objects in the chain do not have to be related at all. The only requirements are that each handler object has the ability to accept, if not actually handle, the request, that it knows which requests it can handle and which it cannot, and that it knows what its successor object is and how to pass the request to it (i.e., what interface to use).

It's easier to see how it works by looking at an object diagram of the Chain of Responsibility pattern in use at runtime, as illustrated in Figure 7.

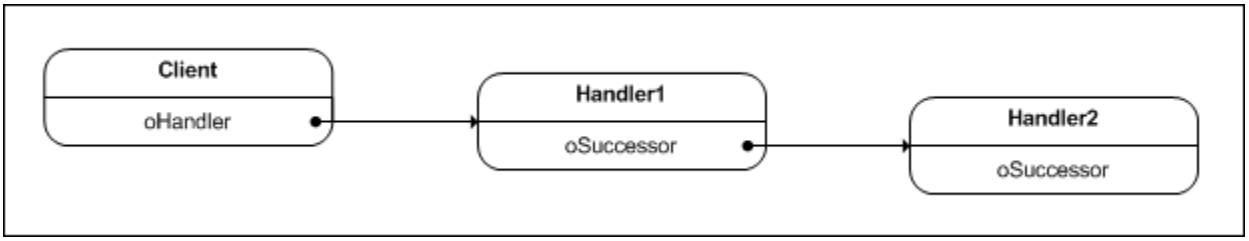

Figure 7: The object structure of the Chain of Responsibility pattern at runtime. This pattern is completely extensible; there is no theoretical limit to the number of handler objects that can be employed.

In a Chain of Responsibility pattern, the Client only needs to be aware of the first handler object in the chain. The Client establishes a "uses a" relationship with that object, either by creating it or by acquiring a reference to it if it already exists. The Client also needs to know

what method to call on the first handler object, but it's up to each of the successor objects to know what their successor object is and what method to call on it.

There is more than one way in which each handler object can know about its successor. One easy way to do it in Visual FoxPro is to store this information in properties of the object. If the first handler object can handle the request, it does so and that's the end of the chain. If the first handler object cannot handle the request, it passes the request on its successor object, first creating that object if necessary. If data is being passed to the hander object in the form of parameters, a handler object than cannot handle the request must pass those parameters on to its successor. If the last handler in the chain cannot handle the request, this creates an exception condition that must be dealt with.

As an example of the Chain of Responsibility pattern, let's return to the Calculate Interest problem we first solved with a Strategy pattern and see how we could implement the solution using a Chain of Responsibility pattern. Listing 8 shows one way of doing this.

Listing 8: The Calculate Interest problem solved with a Chain of Responsibility pattern.
(Session code: \Chain\CalculateInterest_Chain.prg)

```
*--------------------------------------*
*       "Abstraction" Class            *
*--------------------------------------*
DEFINE CLASS myForm as Form
nBalance = 0.00
nRate = 0.00
nPeriod = 0
cType = ""
cHandler = "clsCalculateInterest"
*--------------
FUNCTION GetInterestAmount()
  loHandler = NEWOBJECT( this.cHandler)
  RETURN loHandler.CalculateInterest( this.nBalance, this.nRate, this.nPeriod,
this.cType)
ENDDEFINE

*--------------------------------------*
*       "Handler" Classes              *
*--------------------------------------*
*  Interface class
DEFINE CLASS clsCalculateInterest as custom
cCanHandle = ""
cSuccessor = "clsSimple"
oSuccessor = null
*--------------
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
IF this.cCanHandle = tcType
  *    Virtual
ELSE
  this.oSuccessor = NEWOBJECT( this.cSuccessor)
  RETURN this.oSuccessor.CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
ENDIF
ENDFUNC
```

```
ENDDEFINE

*  Implementation classes (Concrete Handlers)
DEFINE CLASS clsSimple as clsCalculateInterest
cCanHandle = "simple"
cSuccessor = "clsQuarterly"
*--------------
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
IF this.cCanHandle = tcType
  lnInterest = tnBalance * ( ( tnRate / 100) / 365) * tnPeriod
ELSE
  this.oSuccessor = NEWOBJECT( this.cSuccessor)
  lnInterest = this.oSuccessor.CalculateInterest( tnBalance, tnRate, tnPeriod,
tcType)
ENDIF
RETURN lnInterest
ENDFUNC
ENDDEFINE

DEFINE CLASS clsQuarterly as clsCalculateInterest
cCanHandle = "quarterly"
cSuccessor = "clsMonthly"
*--------------
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
IF this.cCanHandle = tcType
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 4)
  ENDFOR
ELSE
  this.oSuccessor = NEWOBJECT( this.cSuccessor)
  lnInterest = this.oSuccessor.CalculateInterest( tnBalance, tnRate, tnPeriod,
tcType)
ENDIF
RETURN lnInterest
ENDFUNC
ENDDEFINE

DEFINE CLASS clsMonthly as clsCalculateInterest
cCanHandle = "monthly"
cSuccessor = "clsDaily"
*--------------
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
IF this.cCanHandle = tcType
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 12)
  ENDFOR
ELSE
  this.oSuccessor = NEWOBJECT( this.cSuccessor)
  lnInterest = this.oSuccessor.CalculateInterest( tnBalance, tnRate, tnPeriod,
tcType)
ENDIF
RETURN lnInterest
ENDFUNC
```

```
ENDDEFINE

DEFINE CLASS clsDaily as clsCalculateInterest
cCanHandle = "daily"
cSuccessor = ""  && no successor
*--------------
FUNCTION CalculateInterest( tnBalance, tnRate, tnPeriod, tcType)
IF this.cCanHandle = tcType
  lnInterest = 0.00
  FOR lni = 1 TO tnPeriod
    lnInterest = lnInterest + ( tnBalance + lnInterest) * ( ( tnRate / 100) / 365)
  ENDFOR
ELSE
  lnInterest = null
ENDIF
RETURN lnInterest
ENDFUNC
ENDDEFINE
```

In this example the handler objects are subclassed from a single parent handler, but again, this is not a requirement. Also, in this example the last handler object is artificially coded to know that it's the last one by returning null if it can't handle the request. For a more generic solution, take a look at \Chain\CalculateInterest_Chain2.prg in the session code.

One potential downside to the Chain of Responsibility pattern is that a relatively large number of objects may need to be created. Another potential issue is garbage collection; the chain should be implemented in such a way as to not to leave unused objects and/or dangling object references around after it's done.

Another example where a Chain of Responsibility pattern could effectively be employed is error handling, where you might want to pass an error along a chain of error handler objects until it reaches the object designed to deal with it in the appropriate fashion. As an example, you might want to implement different handlers for Visual FoxPro errors, OLE errors, and ODBC errors.

The Mediator pattern

The Mediator patterns offers a solution to the problem of coordinating the interaction of separate but related objects without each one having to know about the others. In other words, it enables a group of objects to interact in a desired manner without tightly coupling them to one another. Like Strategy and Chain of Responsibility, Mediator is a behavioral pattern.

Here's the definition of the Mediator pattern along with its class diagram.

> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

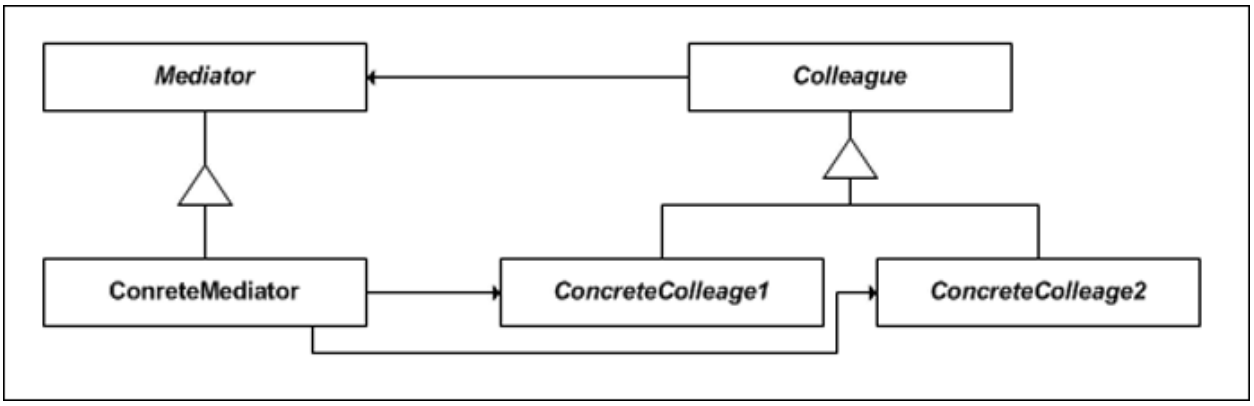

Figure 8: Class diagram of the Mediator pattern.

The Mediator class diagram in Figure 8 is kind of a head-scratcher. As with the Chain of Responsibility pattern, it's easier to see how it works by looking at an object diagram of the pattern at work at runtime.

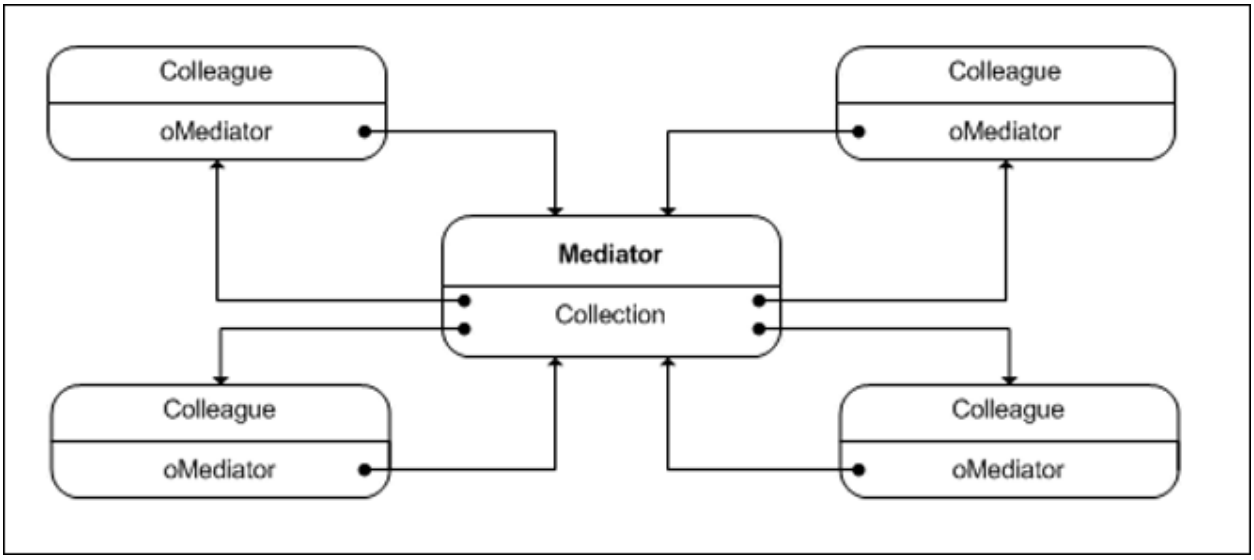

Figure 9: An object diagram showing the Mediator pattern at work at runtime.

Figure 9 shows that the Mediator maintains an object reference to each of its participating objects, called Colleagues. These relationships do not need to be creational; in other words, the Mediator does not need to create the Colleagues, it only needs to have an acquaintance (a "knows a" relationship) with them. The Mediator can maintain the references to its Colleague objects as a collection or in any other way that's convenient. In turn, each of the colleagues maintains an object reference to the Mediator.

The Visual FoxPro OptionGroup control provides a good example of how the Mediator pattern can be used. Although we don't have any way on knowing how this control is internally implemented in VFP, we do know how it behaves. An option group consists of

two or more option buttons. Only one of the option buttons can be selected at any given time, so when a user selects one button the other buttons are automatically unselected.

Because there is no theoretical limit to the number of buttons in an option group, it would clearly be unworkable for each button to communicate directly with all of the others. Instead, each option button only needs to know how to tell the option group that it has changed state (become selected), and the option group takes care of communicating with the other buttons in order to change their state (mark them as unselected) if necessary.

In this example, the option group is the Mediator and the option buttons are the Colleagues. We can easily build some code to demonstrate the mechanics of how this works. For the sake of illustration, the visual appearance is unimportant; all we are concerned with is how the objects interact via the Mediator pattern. Listing 9 illustrates one way this could be done.

Listing 9: The Mediator pattern can be demonstrated using the relationship between an option group and two or more option buttons. (Session code: \Mediator\MediatorPattern.prg)

```
*-------------------------------------*
*         Abstract Mediator class       *
*-------------------------------------*
DEFINE CLASS clsMediator as Custom
cName = ""
*--------------
FUNCTION Init()
this.AddObject( "oColleagues", "collection")
ENDFUNC
*--------------
FUNCTION AddColleague( toColleague)
this.oColleagues.Add( toColleague, toColleague.cName)
ENDFUNC
*--------------
FUNCTION ChangeState( toColleague)
this.UpdateColleagues( toColleague.cName)
ENDFUNC
*--------------
PROTECTED FUNCTION UpdateColleagues( tcName)
FOR EACH loColleague IN this.oColleagues
   IF loColleague.cName = tcName
   ELSE
      loColleague.SetState(.F.)
   ENDIF
ENDFOR
ENDFUNC
ENDDEFINE  && clsMediator

*-------------------------------------*
*         Concrete Mediator               *
*-------------------------------------*
DEFINE CLASS clsOptionGroup AS clsMediator
cName = "OptionGroup"
*--------------
```

```
FUNCTION ShowState()
lcMsg = "There are " + TRANSFORM( this.oColleagues.count) + " buttons." + CHR(13)
FOR EACH loColleague IN this.oColleagues
   lcMsg = lcMsg + loColleague.cName + " " + IIF( loColleague.lSelected, "is
selected", "is not selected") + CHR(13)
ENDFOR
MESSAGEBOX( lcMsg, 0, "Current state of Option Group buttons")
ENDFUNC
ENDDEFINE  && clsOptionGroup

*--------------------------------------*
*         Abstract Colleague class         *
*--------------------------------------*
DEFINE CLASS clsColleague as Custom
cName = ""
oMediator = null
*--------------
FUNCTION Init()
this.oMediator = oMediator  && oMediator is public for demo purposes
ENDFUNC
*--------------
FUNCTION Register()
this.oMediator.AddColleague( this)
ENDFUNC
*--------------
FUNCTION SetState( tlState)
*  Virtual
ENDFUNC
ENDDEFINE  && clsColleague

*--------------------------------------*
*         Concrete Colleague               *
*--------------------------------------*
DEFINE CLASS clsOptionButton as clsColleague
lSelected = .F.
*--------------
FUNCTION Init( tcName)
DODEFAULT()
this.cName = tcName
this.Register()
ENDFUNC
*--------------
FUNCTION Click()
IF this.lSelected
ELSE
  this.SetState( .T.)
  this.oMediator.ChangeState( this)
ENDIF
ENDFUNC
*--------------
FUNCTION SetState( tlState)
this.lSelected = tlState
ENDFUNC
ENDDEFINE  && clsOptionButton
```

In order to help you see how this code works, I've included a driver program to run it and display its state at various points along the way.

Listing 10: Use this driver code to run the Mediator pattern in Listing 9.
(Session code: \Mediator\RunMediatorDemo.prg)

```
PUBLIC oMediator  && For demo purposes, so Colleague objects can see it.
SET PROCEDURE TO mediatorPattern.prg
oMediator = NEWOBJECT( "clsOptionGroup")
*-- Add two buttons
PUBLIC oButton1, oButton2, oButton3, oButton4  && So we can run this code in
fragments.
oButton1 = CREATEOBJECT( "clsOptionButton", "Button1")
oButton2 = CREATEOBJECT( "clsOptionButton", "Button2")
oButton3 = CREATEOBJECT( "clsOptionButton", "Button3")
oButton4 = CREATEOBJECT( "clsOptionButton", "Button4")
oMediator.Showstate()  && No button is selected.

*-- Click button #1
oButton1.Click()
oMediator.Showstate()  && Button 1 is selected.

*-- Click button #3
oButton3.Click()
oMediator.Showstate()  && Button 3 is selected, button 1 is unselected.

*-- Clean up
RELEASE oButton4, oButton3, oButton2, oButton1
RELEASE oMediator
RETURN
```

The Option Group is only one example of where a Mediator pattern can be effectively employed. Another common use is as a Forms Manager in an application framework. In general, the Mediator pattern is applicable anywhere you need to control the state of a group of objects without each object having to communicate explicitly with the others.

The Factory Method pattern

In the real world, factories produce products. In the world of object-oriented software, factories produce objects. The Factory Method pattern provides a way of creating objects when the specific type of object cannot be determined until runtime.

Here's the definition of the Factory Method pattern.

> Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

As you can see from Figure 10, the Factory Method pattern is similar to the Strategy pattern in that it uses subclasses to determine what happens at runtime. The Strategy implements

the required algorithm by passing the request to the appropriate subclass. Similarly, the Factory Method creates the required object by instantiating it from the appropriate subclass. Because it creates an object, the Factory Method is a creational design pattern.

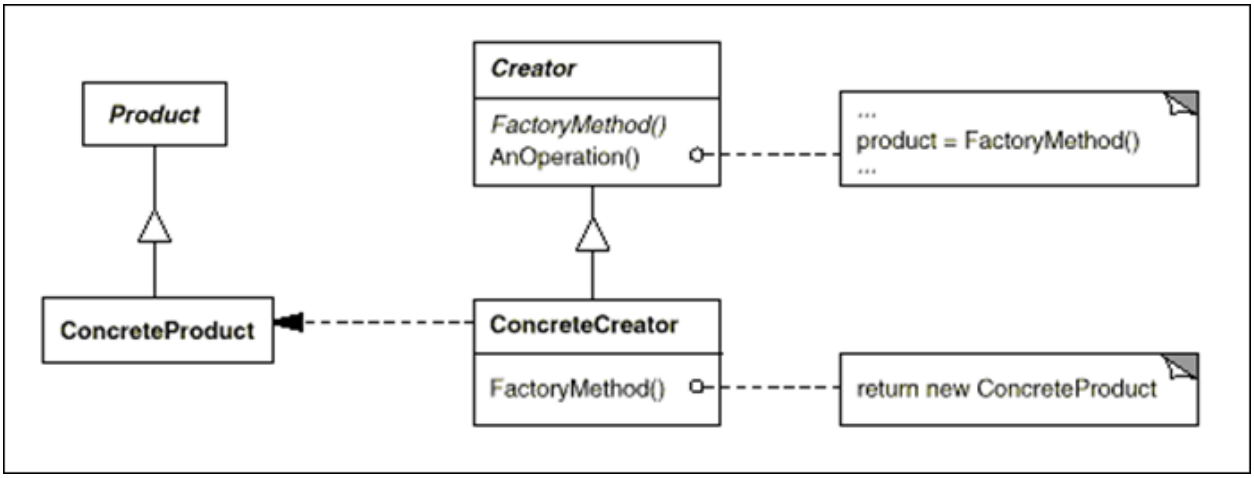

Figure 10: The structure of the Factory Method pattern.

In the Factory Method pattern, the only class with any knowledge of how to actually create the desired object is the Concrete Creator. The job of the Concrete Creator is to create the desired object and return an object reference to the method that invoked it. Because the created object is a subclass of a class with a known interface, the invoking class can then proceed to call methods on the object without needing to know exactly what kind of object it's dealing with.

The FactoryMethod pattern is commonly used in application frameworks because it provides flexibility, enabling the application to instantiate the correct type of object at runtime. As an example, consider a data access class. The class might need to instantiate one kind of data access object if the data is stored in FoxPro tables and a different kind if the data is stored on SQL Server. Another variation could be how the data needs to be returned, for example as a cursor or as XML.

These factors aren't determined by the application framework designers; they're determined in part by the developer who's using the framework to create a specific application and in part by how the application is actually being used in a particular situation. Therefore, the decision about which data access object to create needs to be deferred until runtime.

Let's look at another example of the Factory Method, one that's a little more fun than creating a data access class. If you've been to a café anytime in the last two decades you know how many different variations there are on a basic cup of coffee – plain, latte, mocha, espresso, cappuccino, hot, cold, regular, decaf, with whipped cream or without, with flavored syrup or without, and so on.

We can illustrate the FactoryMethod pattern by designing a software café that can create a cup of coffee. In this example the café is the factory and the coffee is the product. Our café can make two types of drinks, either regular brewed coffee drinks or specialty espresso-based drinks. Which one it creates at any given time is determined by the "runtime" request made by the customer. Listing 11 shows the code to implement this design.

Listing 11: The Café example implements the Factory Method pattern to prepare a cup of the desired coffee and serve it. (Session code: \Factory\Cafe.prg)

```
*--------------------------------------*
*        Creator Classes               *
*--------------------------------------*
DEFINE CLASS Cafe as Custom
oProduct = null
FUNCTION MakeCoffee()
*     Virtual
ENDFUNC
ENDDEFINE

DEFINE CLASS myCafe as Cafe
FUNCTION MakeCoffee( tcType)
this.oProduct = this.GetCoffee( tcType)
this.oProduct.Prepare()
this.oProduct.Serve()
ENDFUNC
*----------
FUNCTION GetCoffee( tcType)
RETURN IIF( tcType = "brewed", NEWOBJECT( "BrewedCoffee"), NEWOBJECT(
"SpecialtyCoffee"))
ENDFUNC
ENDDEFINE

*--------------------------------------*
*        Product Classes               *
*--------------------------------------*
DEFINE CLASS Coffee as Custom
cName = ""
FUNCTION Prepare()
*     Virtual
ENDFUNC
*----------
FUNCTION Serve()
? "Here's your cup of " + this.cName + ". Enjoy!"
ENDFUNC
ENDDEFINE

DEFINE CLASS BrewedCoffee as Coffee
cName = "Brewed Coffee"
FUNCTION Prepare()
? "choose coffee"
? "put in brew machine"
? "add water"
? "brew"
```

```
ENDFUNC
ENDDEFINE

DEFINE CLASS SpecialtyCoffee as Coffee
cName = "Mocha"
FUNCTION prepare()
? "choose coffee"
? "put in espresso machine"
? "pull expresso shot"
? "choose milk"
? "steam milk"
? "add cocoa"
? "mix milk and coffee"
ENDFUNC
ENDDEFINE
```

When a customer orders a cup of coffee, our café creates the correct handler object, displays the steps it takes to prepare the coffee, and serves the drink to the customer. Note that the Prepare and Serve methods are defined in the abstract Product class ("Coffee") and called by the Concrete Creator class ("myCafe").

To run this code from the command line, first create an instance of our café.

```
SET PROCEDURE TO cafe.prg
oCafe = NEWOBJECT( "myCafe")
```

Then, to order a cup of coffee all you have to do is ask the café to serve up the kind you want. For example, you can get a cup of brewed coffee by ordering

```
oCafe.MakeCoffee( "brewed")
```

which returns

```
choose coffee
put in brew machine
add water
brew
Here's your cup of Brewed Coffee. Enjoy!
```

or you can get a cup of mocha by ordering

```
oCafe.MakeCoffee( "mocha")
```

which returns

```
choose coffee
put in espresso machine
pull espresso shot
choose milk
steam milk
```



```
add cocoa
mix milk and coffee
Here's your cup of Mocha. Enjoy!
```

This is an admittedly oversimplified example but it serves to illustrate the concept of the FactoryMethod design pattern. Clearly there are many possible variations within the two categories of "brewed coffee" and "specialty coffee". Think about how you might extend this design, possibly in combination with other design patterns, in order to accommodate some of these other variations.

Using design patterns in combination

When I was a little kid my mother made some of the clothes my brothers and I wore. One of my favorites was a soft, warm corduroy shirt. I watched as she made it, laying out the tissue paper patterns on a table, pinning the material to the patterns, cutting it into the right shapes—body panels, sleeves, cuffs, and collar—and finally sewing the pieces together to make the finished shirt.

I tell this story in order to draw a parallel between how to make a shirt and how to design object-oriented software. When making a shirt, the tissue paper patterns themselves do not become the shirt; they are merely a guide to creating the actual pieces of material that, when combined in the right way by a skilled person, become the finished product. It's the same with designing object-oriented software. Design patterns themselves are not the solutions, they are merely a guide to creating the actual classes that, in the hands of a skilled developer, can be combined to become the finished application.

Summary

The structure of an object-oriented software application at design time is one thing; its structure at runtime is another. At design time, the structure of an object-oriented application is determined by the classes designed by the developer. At runtime, the structure (in computer memory) of the application is determined by the actions of the user and the behavior of the objects created in response to user requests.

The job of the object-oriented software designer is to design the classes and their interactions so that the application behaves in the desired manner, while at the same time maximizing the potential for future changes and enhancements with a minimum of changes to the underlying structure. This is what is meant by the phrase designing for reusability and extensibility.

As you learn design patterns you begin to see object-oriented software design in a different light. This is especially true for long-time FoxPro developers, who, because of FoxPro's procedural roots, may still bring something of a procedural mindset to the task of designing software. Learning to recognize design patterns enables you to approach the software development process with a new and more fully object-oriented perspective, which in turn makes you a better developer.

References

Books

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley. ISBN 0-201-63361-2

Freeman, E. and Freeman, E. 2004. Head First Design Patterns. Sebastopol, CA: O'Reilly Media, Inc. ISBN 978-0-596-00712-6

Shalloway, A., and Trout, J. 2005. Design Patterns Explained, 2nd Ed. Boston, MA: Pearson Education, Inc. ISBN 978-0-321-24714-8

Akins, M., Kramek, A., and Schummer, R. 2002. MegaFox: 1002 Things You Wanted to Know About Extending Visual FoxPro. Whitefish Bay, WI: Hentzenwerke Publishing. ISBN 1-930919-27-1

Printed Articles

Black, S. Design Patterns: Extend and Adapt Your Classes with BRIDGEs. FoxTalk, March 1996. Kent, WA: Pinnacle Publishing, Inc.

Black, S. Design Patterns: Abstract One-to-Many Relationships with OBSERVER. FoxTalk, April 1996. Kent, WA: Pinnacle Publishing, Inc.

Black, S. OOP Design Patterns: Add Design Flexibility with a Strategy Pattern. FoxTalk, January 1997. Kent, WA: Pinnacle Publishing, Inc.

Maskens, S. and Kramek, A. Is it a bird? Is it a plane? No—It's a Pattern! FoxTalk, September 1998. Marrietta, GA: Pinnacle Publishing, Inc.

Donnici, J. Best Practices: Seeing Patterns: The Bridge. FoxTalk, November 1998. Marrietta, GA: Pinnacle Publishing, Inc.

Donnici, J. Best Practices: Seeing Patterns: The Mediator. FoxTalk, December 1998. Marrietta, GA: Pinnacle Publishing, Inc.

Donnici, J. Best Practices: Seeing Patterns: The Observer. FoxTalk, January 1999. Marrietta, GA: Pinnacle Publishing, Inc.

Donnici, J. Best Practices: Seeing Patterns: The Strategy. FoxTalk, February, 1999. Marrietta, GA: Pinnacle Publishing, Inc.

Lassala, C. Implement Design Patterns in Visual FoxPro. Advisor Guide to Microsoft Visual FoxPro, May 2007. San Diego, CA: Advisor Media, Inc.

Conference White Papers

Black, S. Introduction to Object Oriented Design Patterns in Visual FoxPro. Session E-PATT, German DevCon, 1996. http://www.dfpug.de/konf/konf_1996/oop/e_patt/epatt.htm

Kramek, A. Design Patterns in Visual FoxPro. Session E-PATT, 10th European Visual FoxPro DevCon, 2003.

Web References

Siebert, R., Thalacker, P., Testi, A., Fung, D., et al. VFP Design Pattern Catalog. Last updated 2005. Visual FoxPro Wiki. http://fox.wikis.com/wc.dll?Wiki~VFPDesignPatternCatalog

Author(s) unknown. Design Patterns. http://www.oodesign.com/

