CenturaPro™
www.ProPublishing.com
Hot Ideas for Centura® Developers

Try *This!*

Johan Alm

April 2000
Volume 5, Number 4

PRO Publishing

OK, buckle your seat belts. The author begins by exploring a simple design concept and ends up with a dizzying array of design patterns, all interlocking and supporting each other. Come see how the imminent release of CTD 2000 puts new life into object-oriented programming in SAL!

For this example, let's look at the mother of all data structure examples: the stack. **Listing 1** (see page 3) shows a simple implementation of a stack with a maximum capacity, a function to get the capacity, as well as the standard Pop, Push, and Peek operations. You can query the stack about the number of elements in it. There are also three "private" versions of Pop, Peek, and Push that all begin with a double underscore. In this way Pop, Peek, and Push just determine if the action is legal, and if so let the corresponding private function do the real work. This separation of concern might seem like overkill in this simple example, and, in fact, it is, but as we'll see later, this helps us modify the stack in different ways.

SQLWindows
32

The only other thing outside the ordinary is that the Push function will only accept non negative numbers, since -1 is used as a return value from Pop and Peek to indicate an error, for example, trying to Pop an empty stack.

The reason I use the stack for this example is simply because it exhibits state-dependent behavior. The functions Pop, Push, and Peek all behave differently depending on the state of the stack, in other words, if the stack is empty, full, or somewhere in between. The stack is really a little bit too simple—both in terms of the number of states and the number of state-dependent functions and their complexity—to fully illustrate the topics I'll discuss in this article. Just imagine a class with many states and many complex state-dependent functions as soon as the stack is mentioned, and you'll get the idea.

State-dependent functions typically contain conditional statements, often with as many branches as the object has states. This is sometimes a problem for several reasons:

- If a new state has to be introduced or an existing one removed, all state-dependent functions must be changed.
- If there are many states, the code for the state-dependent functions can become large, with many conditional branches—and thus hard to understand and maintain.
- The state of an object is often captured by an instance variable (for example, if inCount = 0 the stack is empty) and sometimes several, for instance, the stack is full if inCount = inCapacity. It can therefore be hard to understand from the code what different states the object may have.

Continues on page 3

Time To Go

Mark Hunter

This is the last issue of *Centura Pro*. (No, I'm not making an April Fool joke.)

The principals of Pro Publishing are moving on to other, long-postponed objectives. Dian Schaffhauser, the publisher, is returning to university studies. I'm taking two months off to travel across North America with my son and daughter. They're nine and ten years old, and I want to make this big trip before they're teenagers who can't stand spending time with me!

When we started *Gupta Pro* more than four years ago (before Centura Software changed its name), many things were different. There was no 32-bit version of the development tools. Centura's stock price was about five dollars, headed for a long slump. The World Wide Web was a moderately popular phenomenon, and some companies had built uncomplicated Web sites. Intranets were virtually nonexistent.

Now Centura stock is volatile but healthy. 32-bit Centura Team Developer is about to become much more robust and Web-friendly with the release of CTD 2000. Sophisticated Web sites and intranets are now mandatory, and software developers must know how to support them.

Over the years, one thing didn't change. Pro Publishing had the strong support of its readers and authors. Through some difficult times, you provided the articles and the market that kept this newsletter so valuable. That same spirit makes the Centura newsgroups lively and useful and drives the innovations of third-party developers. You are fine people, and all of us at Pro Publishing express our deep thanks for being welcomed as part of your community.

The newsletter is ending, but the Centura community continues to move forward. As evidence of this, read Johan Alm's intensive look at new OOP techniques in CTD, in this issue. I will continue to work as a Centura developer, and I look forward to seeing you in newsgroups and conferences. Perhaps, with some spare time, I can even write an informal article or two for posting on a Web site somewhere! We work with a tool that is too good to disappear, and I think its darkest hours are now behind us. Take CTD onward, and build great apps with it.

This has sometimes been a tough job for me, but it has always been rewarding and interesting. Once again, thank you for allowing me to serve you. **CP**

Enditor Mark Hunter, **Scrublisher** Dian Schaffhauser, **Going out of Business Manager** Shelley Doyle, **Production Enditor** Paul Gould, **Centura Dog (Her Bark is Worse Than Her Bye)** Mocha

Centura Pro (ISSN: 1093-2100) is published monthly (12 times per year) by Pro Publishing, PO Box 2399, Nevada City, CA 95959.

POSTMASTER: Send address changes to *Centura Pro*, PO Box 2399, Nevada City, CA 95959.

Contact Us

Centura Pro on the Web
www.ProPublishing.com

Editorial Department
Phone: 818-249-1364
Fax: 818-246-0487
E-mail: huntersoftware@netscape.net

Subscription Services
Phone: 530-265-4082
Fax: 530-265-0368
E-mail: shelley@propublishing.com

Mail
Pro Publishing
PO Box 2399
Nevada City, CA 95959

Try *This!*

Continued from page 1

Listing 1. A simple stack implementation.

```
◆ Functional Class: CSimpleStack
  ◆ Instance Variables
    ◇ Number: theStack[*]
    ◇ Number: inCapacity
    ◇ Number: inCount
  ◆ Functions
    ◆ Function: Push
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ Number: pnNumber
      ◆ Actions
        ◆ If pnNumber < 0
          ◇ Return FALSE
        ◆ Else
          ◆ If inCount < inCapacity
            ◇ Call __Push(pnNumber)
            ◇ Return TRUE
          ◆ Else
            ◇ Return FALSE
    ◆ Function: Pop
      ◆ Returns
        ◇ Number:
      ◆ Local variables
        ◇ Number: n
      ◆ Actions
        ◆ If inCount > 0
          ◇ Set n = __Pop()
        ◆ Else
          ◇ Set n = -1
        ◇ Return n
    ◆ Function: Peek
      ◆ Returns
        ◇ Number:
      ◆ Local variables
        ◇ Number: n
      ◆ Actions
        ◆ If inCount > 0
          ◇ Set n = __Peek()
        ◆ Else
          ◇ Set n = -1
        ◇ Return n
    ◆ Function: GetCapacity
      ◆ Returns
        ◇ Number:
      ◆ Actions
        ◇ Return inCapacity
    ◆ Function: __Push
      ◆ Returns
        ◇ Number:
      ◆ Parameters
        ◇ Number: pnNumber
      ◆ Actions
        ◇ Set theStack[inCount] = pnNumber
        ◇ Set inCount = inCount + 1
        ◇ Return inCount
    ◆ Function: __Pop
      ◆ Returns
        ◇ Number:
      ◆ Actions
        ◇ Set inCount = inCount - 1
        ◇ Return theStack[inCount]
    ◆ Function: __Peek
      ◆ Returns
        ◇ Number:
      ◆ Actions
        ◇ Return theStack[inCount-1]
    ◆ Function: GetCount
      ◆ Returns
        ◇ Number:
      ◆ Actions
        ◇ Return inCount
```

State of the nation

The State pattern will show us an alternative way to handle state-dependent behavior. The basic idea is to make the state of the stack into a class of its own, CStackState. Every possible state of the stack is implemented as a subclass to CStackState. The stack itself will hold a reference to a CStackState object that represents the current state of the stack. Changing the state of the stack means changing the state *object* of the stack. The state-dependent functions in the stack are changed so that they do no work of their own but instead call the corresponding function on the current state object. By the way, this behavior is often referred to as delegation in object-oriented literature. The late-bound mechanism will then see to it that the appropriate state-dependent function is called depending on the actual class of the current state object. **Figure 1** shows the class structure when implementing the stack using the state pattern.

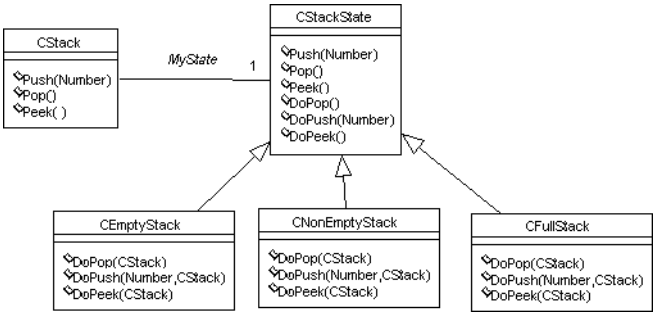

Figure 1. The State pattern.

Let's apply this pattern on the stack. We need to add an instance UDV, MyState, to hold the reference to the current state of the stack. Next we change the state-dependent function of the stack so that they just call the corresponding function on current state object. But wait a second, how is the state object to know what stack to modify? One solution is to have every state object permanently referring to its stack by means of an instance UDV, but this would imply that state objects can't be shared between stacks. If you think about it, there's no reason we couldn't share state objects. They hold no data, only behavior; so several stacks being in the same state could share a common state object. If we have many stacks, we can get away with only a few shared state objects, but then the state object can't hold a reference to its stack, as outlined above.

Get hold of yourself!

Fortunately, there is a new feature to solve this problem: the **this** reference, which is for functional classes what hWndItem is for Windows-based classes, a self reference. We just have to add a second parameter to the state-dependent functions in the state class, namely the stack to be modified. Since it's the stack making the call to the state object, it passes itself as a parameter using the keyword "this." At last, our UDVs are aware of themselves!

The state-independent functions like GetCapacity haven't been moved to the state object, but remain in the stack. **Listing 2** shows the new version of the CStack class.

While we're at it, we'll also change the stack to accept any objects and not just numbers. This is simply a matter of changing the type of the array holding the stack elements to CObject and changing the parameter of Push and return values of Pop and Peek to the same type. Any object derived from CObject can now be handled by the stack. Not only can different stack objects handle different kinds of information, but one stack can also contain elements of different types.

Listing 2. The stack with modified state-dependent functions and changed to accept any objects.

```
◆ Functional Class: CStack
  ◆ Instance Variables
    ◇ CStackState: MyState
    ◇ CObject: theStack[*]
  ◆ Functions
    ◆ Function: Push
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObject: pObj
      ◆ Actions
          ◇ Return MyState.Push(pObj,this)
    ◆ Function: Pop
      ◆ Returns
        ◇ CObject:
      ◆ Actions
        ◇ Return MyState.Pop(this)
    ◆ Function: Peek
      ◆ Returns
        ◇ CObject:
      ◆ Actions
        ◇ Return MyState.Peek(this)
    ◆ Function: __Push
      ◆ Parameters
        ◇ CObject: pObj
      ◆ Actions
        ◇ Set theStack[inCount] = pObj
        ◇ Set inCount = inCount + 1
    ◆ Function: __Pop
      ◆ Returns
        ◇ CObject:
      ◆ Actions
        ◇ Set inCount = inCount - 1
        ◇ Return theStack[inCount]
    ◆ Function: __Peek
      ◆ Returns
        ◇ CObject:
      ◆ Actions
        ◇ Return theStack[inCount]
```

From Listing 2 we understand that all states must be able to handle the Pop, Peek, and Push operations. So these are added to the base class CstackState, and late-bound calls ensure that the appropriate subclass function is called. **Listing 3** shows the abstract CStackState class. Notice that the push operation checks if the parameter is a null object and, if so, rejects the call. This logic serves the same purpose as the one that checked to ensure that the number to push was positive and is required, since we now use OBJ_Null to indicate failure. The default implementation of the late-bound functions DoPop and DoPeek returns a null reference, and Push returns FALSE. This is necessary since a stack will start its life having MyState referring to an object of this base class. Since we didn't declare it to be initialized, its value will be OBJ_Null. As long as the capacity of the stack hasn't been set, the stack is useless, since all the calls will be delegated to the state base class functions, which will return a null object or FALSE.

Since I don't intend to let the capacity be changed after the creation of the stack, I didn't supply a SetCapacity function. Instead, I use the approach I introduced in my last article: using the "fake" constructor function in **Listing 4** to create stacks. This one will set the instance variable directly and, since I view this function as an integrated part of the class, I will let it do so. Also, introducing a SetCapacity function could lead a user of the stack to erroneously believe that the capacity can be changed. Finally, notice how the "constructor" ensures that the new stack starts off in the empty stack state. This is done by calling the __SetState( ) function, which is just an encapsulation of assigning MyState to the new state object.

Listing 3. The CStackState class.

```
◆ Functional Class: CStackState
  ◆ Functions
    ◆ Function: Pop
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: theStack
      ◆ Actions
        ◇ Return ..DoPop(theStack)
    ◆ Function: Peek
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: theStack
      ◆ Actions
        ◇ Return ..DoPeek(theStack)
    ◆ Function: DoPop
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: theStack
      ◆ Actions
        ◇ Return OBJ_Null
    ◆ Function: DoPeek
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack  : theStack
      ◆ Actions
        ◇ Return OBJ_Null
    ◆ Function: Push
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObject: pObj
        ◇ CStack: theStack
      ◆ Actions
        ◆ if pObj != OBJ_Null
          ◇ Return ..DoPush( pObj, theStack)
        ◆ else
          ◇ Return OBJ_Null
    ◆ Function: DoPush
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObject: pObj
        ◇ CStack  : theStack
      ◆ Actions
        ◇ Return FALSE
```

Listing 4. Pseudo constructor for CStack.

```
◆ Function: CreateStack
  ◆ Returns
    ◇ CStack:
  ◆ Parameters
    ◇ Number: pnCapacity
  ◆ Local variables
    ◇ CStack: aNewStack
  ◆ Actions
    ◇ Set aNewStack.inCapacity = pnCapacity
    ◇ Call aNewStack.__SetState( new CEmptyStack )
    ◇ Return aNewStack
```

Let's look at one concrete subclass of CStackState, the CEmptyState as shown in **Listing 5**. It implements the late-bound calls from the base class, and there are two things worth noticing about the implementation. Let's use the Push function as an example even though the same applies to all three of the state-dependent functions.

The value to push onto the stack is supplied as a parameter, and so is the stack. The CEmptyStack could manipulate the instance variables of the stack directly in order to push the value, but for various reasons (my being an OO purist is one) I've chosen to use the "private" __Push( ) function of the stack (I told you we'd find them useful, didn't I?). In C++ the functions __Pop( ), __Push( ) and __Peek( ) could have been declared as private and the CStackState class declared as a friend to the CStack.

The other issue involves the state changes. In this case we push something onto an empty stack, which should lead to the stack changing its state. Using the state pattern, this means that the stack should now refer to another state object—an object of the class CNonEmptyStack. There are at least two different approaches to this; let the stack change its own state or let the current state object change the stack state when necessary. The first approach makes the state classes completely independent of each other but makes the stack aware of all the subclasses of CStackState. The other makes the stack independent of the concrete state objects but makes the state object aware of at least one of the others. In this example I'll let the state object handle the state transition of the stack.

The CEmptyStack class "knows" that the stack should change state when a push occurs but also that the state to change to is CNonEmptyStack since the current state is an empty stack. Therefore we conclude the DoPush function by creating a new CNonEmptyStack and telling the stack to use that state object from now on. This is made by calling SetState( ). As seen from Listing 5 some of the late-bound functions don't have to be implemented in every subclass but can use the default implementation in the base class.

Listing 5. The subclasses of CstackState.

```
◆ Functional Class: CEmptyStack
  ◆ Derived From
    ◇ Class: CStackState
  ◆ Functions
    ◆ Function: DoPush
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObject: pObj
        ◇ CStack: pStack
      ◆ Actions
        ◇ Call pStack.__Push(pObj)
        ! Change state, the stack is no longer empty
        ◇ Call pStack.__SetState( new CNonEmptyStack )
        ◇ Return TRUE

◆ Functional Class: CNonEmptyStack
  ◆ Derived From
    ◇ Class: CStackState
  ◆ Functions
    ◆ Function: DoPop
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: pStack
      ◆ Local variables
        ◇ CObject: temp = OBJ_Null
      ◆ Actions
        ◇ Set temp = pStack.__Pop()
        ◆ If pStack.GetCount() = 0
          ◇! Pop of the last element, we now
            ! have an empty stack
          ◇ Call pStack.__SetState( new CEmptyStack)
        ◇ Return temp
    ◆ Function: DoPeek
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: pStack
      ◆ Actions
        ◇ Return pStack.__Peek()
    ◆ Function: DoPush
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObject: pObj
        ◇ CStack: pStack
      ◆ Actions
        ◇ Call __Push(pObj)
        ◆ If pStack.GetCount() = pStack.GetCapacity()
          ◇! Reached the max capacity,
            ! we now have a full stack
          ◇ Call pStack.__SetState( new CFullStack)
        ◇ Return TRUE

◆ Functional Class: CFullStack
  ◆ Derived From
    ◇ Class: CStackState
  ◆ Functions
    ◆ Function: DoPop
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: pStack
      ◆ Local variables
        ◇ CObject: temp = OBJ_Null
      ◆ Actions
        ◇ Set temp = pStack.__Pop()
        ◇! The stack is no longer full.
        ◇ Call pStack.__SetState( new CNonEmptyStack )
        ◇ Return temp
    ◆ Function: DoPeek
      ◆ Returns
        ◇ CObject:
      ◆ Parameters
        ◇ CStack: pStack
      ◆ Actions
        ◇ Return pStack.__Peek()
```

Wrapper classes and the Flyweight pattern

Our stack is very general, in the sense that any class derived from CObject can be handled; but what about primitive SAL types like Number? The only way to get these into the stack is to wrap them in a class like CNumber. This wrapper class contains an instance variable to hold the actual value and a function to get the value as a number out of the object. **Listing 6** shows a simple wrapper class for the Number type.

Listing 6. A wrapper class for numbers.

```
◆ Functional Class: CNumber
  ◆ Derived From
    ◇ Class: CObject
  ◆ Instance Variables
    ◇ Number: theValue
  ◆ Functions
    ◆ Function: GetValue
      ◆ Returns
        ◇ Number:
      ◆ Actions
        ◇ Return theValue
  ◆ Function: GetAsStr
      ◆ Returns
        ◇ String:
      ◆ Actions
        ◇ Return SalNumberToStrX(theValue,0)
```

Using objects from this wrapper class can be a problem, performance-wise. We might end up with a large number of such objects, especially if we use this class for all numbers, and not just the ones that have to go into a stack. The Flyweight pattern offers a remedy for this, provided that the CNumber objects are immutable (in other words, can't be changed). We don't need a new CNumber object for every number; objects representing the same number can be shared, so we only need one CNumber object for every distinct number. To do this we must create the CNumber object through the CreateNumber function shown in **Listing 7**. When the caller requests a new number object, this function will check to see whether that number has been requested before. If not, a new object is created and stored within the function before being returned. If the number has, indeed, been used before, the corresponding object is returned. This has given the name to the pattern; CNumber objects are low-cost flyweight objects.

Listing 7. Creating Flyweight CNumber objects.

```
◆ Function: CreateNumber
  ◆ Returns
    ◇ CNumber:
  ◆ Parameters
    ◇ Number: pnValue
  ◆ Static Variables
    ◇ Number: inCount
    ◇ Number: inaValues[*]
    ◇ CNumber: theNumberObjects[*]
  ◆ Local variables
    ◇ Number: nIndex
  ◆ Actions
    ◇ Set nIndex= VisArrayFindNumber(pnValue,inaValues)
    ◆ If nIndex < 0
      ◇ Set nIndex = inCount
      ◇ Set theNumbers[inCount] = new CNumber
      ◇ Set theNumbers[inCount].theValue = pnValue
      ◇ Set inaValues[inCount] = pnValue
      ◇ Set inCount = inCount + 1
    ◇ Return theNumberObjects[nIndex]
```

Before we continue, I should clarify that the flyweight pattern could be applied to state objects as well. After all, state objects hold no data, which makes them good candidates for being implemented as flyweight objects.

The only difference would be that there are a number of state classes as opposed to only one number class. **Listing 8** shows a possible implementation of this. The implementation of CreateState demonstrates a few new CTD 2000 features. First, the supplied state name parameter is checked to see if it's the name of an existing class through the SalObjIsValidClassName function. This function returns true if the parameter is the name of an existing class, otherwise false. Following that, we have three If statements, and all of them show different ways of creating a new object. The first uses the new SalObjCreateFromString function; the second uses the new keyword in the "normal" way; and the third uses the new keyword with a string as a parameter. This third case is just another way of creating an object from a string, so SalObjCreateFromString is really redundant.

Listing 8. The Stack State Object Factory.

```
◆ Functional Class: CStackStateFactory
  ◆ Instance Variables
    ◇ CFullStack: fullStac = OBJ_Null
    ◇ CEmptyStack: emptyStack = OBJ_Null
    ◇ CNonEmptyStack: nonEmptyStack = OBJ_Null
  ◆ Functions
    ◆ Function: CreateState
      ◆ Returns
        ◇ CStackState:
      ◆ Parameters
        ◇ String: psStateName
      ◆ Local variables
        ◇ CStackState: aNewState = OBJ_Null
      ◆ Actions
        ◆ If SalObjIsValidClassName(psStateName)
          ◆ If psStateName = "CFullStack"
            ◆ If fullStack = OBJ_Null
              ◇ Set fullStack =
                  SalObjCreateFromString(psStateName)
            ◇ Set aNewState = fullStack
          ◆ If psStateName = "CNonEmptyStack"
            ◆ If nonEmptyStackState = OBJ_Null
              ◇ Set nonEmptyStack= new CNonEmptyStack
            Set aNewState = nonEmptyStack
          ◆ If psStateName = "CEmptyStack"
            ◆ If emptyStack = OBJ_Null
              ◇ Set emptyStack= new psStateName
            ◇ Set aNewState = emptyStacks
          ◇ Return aNewState
        ◆ Else
          Return OBJ_Null
```

The only thing that remains is to modify the functions in the state classes that perform state transitions to make use of the CStackStateFactory class, and to make an object of that class available to them. There must be only one instance of the factory class. **Listing 9** shows one example of how a modified state class might look.

Listing 9. The DoPop function of the CEmptyStack class modified to use the state factory.

```
◆ Function: DoPop
  ◆ Returns
    ◇ Number:
  ◆ Parameters
    ◇ CStack: pStack
  ◆ Local variables
    ◇ Number: n
  ◆ Actions
    ◇ Set n = pStack.__Pop()
    ◆ If pStack.GetCount() = 0
      ◆ Call pStack.__SetState(Factory.CreateState
        ("CEmptyStack" ))
    ◇ Return n
```

Command and Conquer

Let's add a UI to our stack. It could look like the one in **Figure 2**. Before we start writing the code for this, let's take a step back and contemplate the situation. We have three distinct commands (Pop, Peek, and Push) that we want to trigger from a number of places in the UI. Not only do we want to trigger the Pop command from the push button labeled Pop, we also want to do the same thing through a menu item, through a context menu in the list box showing the content of the stack, and so on. This means we have one command but a number of different ways to trigger it. Enter the Command pattern.

The key to this pattern is the notion of a command as an object in its own right. It has at least one function, Execute, and typically contains information about its

target, for example, a stack for commands relating to stacks. In this way the trigger (button) is de-coupled from the target (the stack) and the two are held together by a command object. A button can be given a command without knowing anything about it except that it should call the command's Execute function upon SAM_Click, and the command object will then know what to do. The Command pattern applied to our stack example gives the structure in **Figure 3**.

Now we can write a general-purpose push button class like the one in **Listing 10**. Every such button has a command object, initially set to OBJ_Null. Someone, be it the button itself on SAM_Create or some other class, will create a command and give it to the button through the SetCommand function. On SAM_Click the class will call the Execute function on its command if such exists. Notice that the Execute command takes no parameters. After all, it's something that all commands must be able to do; but some—like Pop and Peek—require no parameters and some—like Push—do. The same goes for return values; some commands return a value, some don't.

Listing 10. The CcommandButton.

```
◆ Pushbutton Class: CCommandButton
  ◆ Instance Variables
    CStackCommand: MyCommand = OBJ_Null
  ◆ Functions
    ◆ Function: SetCommand
      ◆ Parameters
        ◇ CStackCommand: pCommand
      ◆ Actions
        ◇Set MyCommand = pCommand
  ◆ Message Actions
    ◆ On SAM_Click
      ◆ if MyCommand != OBJ_Null
        ◇ Call MyCommand.Execute()
```

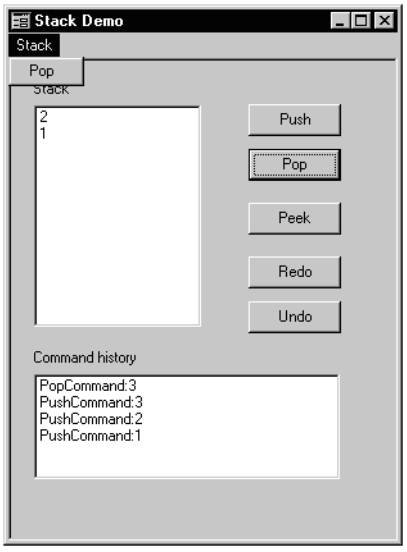

Figure 2. The user interface.

The CStackCommand is a base class for all command classes pertaining to a stack. It has a UDV of the CStack class so that every command object knows the stack that the command is to be executed upon (known as the command target). It has a CObject instance variable used by the Push Command to get the object to push into the stack. This UDV is also used by the Pop and Peek commands to store the result of the stack operation. Therefore, the users of command objects must call the SetValue function prior to Execute of a Push command and use GetValue to retrieve the result of executing a Pop or Peek ditto.

This implies that certain conditions have to be fulfilled before a command can be executed, and this is handled by the IsExecutable function shown in **Listing 11**. This function is used within the class, but it can also be used from the outside. Common to all stack commands is the requirement that the target stack must have been set. There might be additional requirements by the different subclasses, which is solved by the late-bound call to DoIsExecutable. The only subclass that makes an implementation of this in my example is the push command, since it must also have something to push.

Listing 11. The CStackCommand base class.

```
◆ Functional Class: CStackCommand
  ◆ Derived From
    ◇ Class: CObject
  ◆ Instance Variables
    ◇ CObject: theValue = OBJ_Null
    ◇ CStack: MyStack = OBJ_Null
  ◆ Functions
    ◆ Function: SetValue
      ◆ Parameters
        ◇ CObject: pObj
      ◆ Actions
        ◇ Set theValue = pObj
    ◆ Function: GetValue
      ◆ Returns
        ◇ CObject:
      ◆ Actions
        ◇ Return theValue
    ◆ Function: Execute
      ◆ Actions
        ◆ if IsExecutable()
          ◇ Call ..DoExecute()
    ◆ Function: DoExecute
      ◆ Actions
        ◇! No action
    ◆ Function: IsExecutable
      ◆ Returns
        ◇ Boolean:
      ◆ Actions
        ◇ Return ( MyStack !=
          OBJ_Null and ..DoIsExecutable())
      ◆ Function: DoIsExecutable
        ◆ Returns
          ◇ Boolean:
        ◆ Actions
          ◇ Return TRUE
```

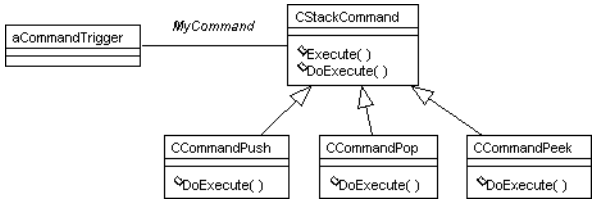

Figure 3. The Command pattern.

In **Listing 12** ,one specific command subclass is shown, the CCommandPush. Not a very complicated one, indeed. Some of you might wonder why I introduce all these classes to handle something that could just as well have been solved with a few lines of code under SAM_Click? The answer is that once we have the Command pattern in place, we can do a lot of other neat stuff, as you'll see.

Listing 12. The command class for push operations.

```
◆ Functional Class: CCommandPush
  ◆ Derived From
    ◇ Class: CStackCommand
  ◆ Functions
    ◆ Function: DoExecute
      ◆ Actions
        ◇ Call MyStack.Push(theValue)
    ◆ Function: DoIsExecutable
        ◆ Returns
          ◇ Boolean:
        ◆ Actions
          ◇ Return theValue != OBJ_Null
```

If you change your mind . . .

How about undo/redo on all operations in the stack? This just requires two additional things: First, keep track of the commands that have been executed and, second, introduce an Undo function on the command class. In the CStackCommand the Undo function (without parameters or return value) just makes a late-bound call to DoUnExecute. **Listing 13** shows this function in the CCommandPush subclass. To undo a Push, we just have to create a Pop command and execute it. Once again we use a global function returning a UDV to simulate a constructor. Since the Pop must be performed on the same stack as the Push command we want to undo, this stack is passed as a parameter to the "constructor." Finally, notice that it's possible to call the Execute function directly on the return value of the CreateCommand function, freeing us from the usage of temporary variables. The CCommandPop class is extended in the same way, but the CCommandPeek doesn't have to implement its own version of DoUnExecute since the default one in the base class will do. After all, there's not much to undo from a peek command.

Listing 13. The DoUnExecute function added to CCommandPush.

```
◆ Function: DoUnExecute
  ◆ Actions
    ◇ Call CreateCommand
      ("CcommandPop", MyStack).Execute()
```

Listing 14. The pseudo constructor for creating command objects.

```
◆ Function: CreateCommand
  ◆ Returns
    ◇ CStackCommand:
  ◆ Parameters
    ◇ String: psCommandName
    ◇ CStack: pTarget
  ◆ Local variables
    ◇ CStackCommand: newCommand = OBJ_Null
  ◆ Actions
    ◆ If SalObjIsValidClassName(psCommandName)
      ◇ Set newCommand = new psCommandName
      ◇ Set newCommand.theTargetStack = pTarget
    ◇ Return newCommand
```

What remains now is the following: Every executed command must be stored so that it can be undone/redone at a later point. For some purposes a simple reference to the last executed command will suffice to achieve undo/redo of the last command. To get multi-level undo/redo, some kind of structure to hold executed commands must be implemented. We also need to keep track of the last command inserted into the structure.

If a command is to be undoable or redoable, we must see to it that it's not changed after it has been executed. Therefore, we'll require the ability of every command to make a copy of itself. We'll then store a copy of the executed command for redo/undo purposes.

Go clone yourself—again!

Let's start with the last item, the Clone function. This is partly implemented in the Clone function of CStackCommand and partly by the late-bound DoClone functions in the subclasses. **Listing 15** shows the new Clone function in the base class. The function SalObjGetType is another addition to the OOP arsenal in CTD 2000. It returns the name of the class referenced by the parameter and, by using that name, we'll create a new object of the appropriate class. After that we just copy the instance variables to the new object. Finally, there's a late-bound call to the subclass passing the new object. This is necessary since there might be instance variables in the subclass that need to get copied to the new object. However, in our simple example there's no instance variable to copy in the subclasses.

Listing 15. The Clone function in CStackCommand.

```
◆ Function: Clone
  ◆ Returns
    ◇ CStackCommand:
  ◆ Local variables
    ◇ CStackCommand: aNewCommand = OBJ_Null
  ◆ Actions
    ◇ Set aNewCommand = new SalObjGetType(this)
    ◇ Call aNewCommand.SetValue( GetValue() )
    ◇ Set aNewCommand.MyStack = MyStack
    ◇ Call ..DoClone(aNewCommand)
    ◇ Return aNewCommand
```

The question about where to store the executed commands is solved by yet another stack, the CCommandHistory shown in **Listing 16**. It's derived from CStack but it also introduces two new functions: Redo and Undo. Every executed command will be pushed onto the stack and Redo will take the topmost command, clone it, execute it and push it onto the stack. (Shortly we'll see that by calling Execute, we'll have a copy of the executed command pushed to the CCommandHistory; all we really have to do is call Execute on the topmost command.)

Undo pops the stack and calls Undo on the resulting command object. (Once again, since every executed command will be pushed to the stack and, undo of a push will create and execute a Pop command, we'll actually need a second pop on the CCommandHistory to get rid of the Pop.) Even though CStack can hold a mix of different kinds of objects, we want to make sure that only command objects can be pushed onto this stack. We achieve this by overriding the Push function and declaring the parameter as CStackCommand. While we're at it, we'll override Peek and Pop as well, changing the return value to CStackCommand. This makes it possible to write very compact versions of Undo and Redo. If we hadn't overridden Peek and Pop, we couldn't have called Execute directly on the result of Peek in the Redo function; the return value would have been CObject and that class knows nothing of such functions as Execute. Instead, we would have needed a temporary variable of the type CStackCommand to hold the result of the Peek call (making a type cast) and then call Execute in the temporary variable.

Listing 16. Redo and Undo in CCommandHistory.

```
◆ Functional Class: CCommandHistory
  ◆ Derived From
    ◇ Class: CStack
  ◆ Functions
    ◆ Function: Push
    ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CStackCommand: pObj
      ◆ Actions
          ◆ if SalObjGetType(pObj)
            ◇ Return CStack.Push(pObj,this)
          ◆ else
            ◇ Return FALSE
    ◆ Function: Pop
      ◆ Returns
        ◇ CObject:
      ◆ Actions
          ◇ Return CStack.Pop(pObj,this)
    ◆ Function: Peek
      ◆ Returns
        ◇ CObject:
      ◆ Actions
          ◇ Return CStack.Peek(pObj,this)
    ◆ Function: Redo
      ◆ Actions
        ◇ Call Peek().Execute()
        ◇ Call Push(Peek().Clone())
    ◆ Function: Undo
      ◆ Actions
        ◇ Call Pop().Undo()
```

The only problem left is to decide who puts the executed commands in the CCommandHistory stack. It could be done by whoever calls the Execute function on a command object; but that's done in many places, and it's easy to forget to push the executed command to the history stack right after the execute. This also requires every client to have access to the history stack. My preferred method is to solve this within the Execute function of the CStackCommand as in Listing 17. Since every call to Execute passes through this base class function before ending up in a late-bound function in one of the subclasses, we can never forget to push the command object to the CCommandHistory. Now let every stack contain a CCommandHistory UDV to allow a command history for each individual stack. **Listing 18** shows the additions to CStack. The initialization of MyHistory to OBJ_Null is required. Without it we would get a "Circularly defined class" error message, because CCommandHistory is derived from CStack but now we've added a CCommandHistory UDV as an instance variable to CStack. To avoid using such a null reference, we introduce the GetHistory( ) function, since it will create a new CCommandHistory if necessary. Of course, this implies that we must always use this function to access the instance variable MyHistory, even from within the class itself.

Listing 17. Augmenting Execute in CStackCommand.

```
◆ Function: Execute
      ◆ Actions
        ◆ Call ..DoExecute()
        ◆ Call MyStack.AddToHistory(this)
```

Listing 18. Adding command history to CStack.

```
◆ Functional Class: CCommandStack
  ◆ Instance Variables
    ◇ CCommandHistory: MyHistory = OBJ_Null
  ◆ Functions
    ◆ Function: AddToHistory
      ◆ Parameters
        ◇ CStackCommand: pCommand
      ◆ Actions
        ◇ Call GetHistory().Push(pCommand)
    ◆ Function: GetHistory
      ◆ Returns
        ◇ CCommandStack:
      ◆ Actions
          ◆ If MyHistory = OBJ_Null
            ◇ Set MyHistory = CreateCommandStack(100)
          ◇ Return MyHistory
    ◆ Function: RedoLastCommand
      ◆ Actions
        ◇ Call MyHistory.Redo()
    ◆ Function: UndoLastCommand
      ◆ Actions
        ◇ Call MyHistory.Undo()
```

The return of the Composite pattern

If you read my article in the March 2000 issue of *Centura Pro*, you might remember the Composite pattern. If we combine the Composite pattern with the Command pattern, we get a macro. A macro is made up of a number of commands, is a subclass of CStackCommand, and thus supports the same interface and can be used anywhere a simple command can be used. **Listing 19** shows the code for CMacro.

Listing 19. The interesting parts of the CMacro class.

```
◆ Functional Class: CStackMacro
  ◆ Derived From
    ◇ Class: CStackCommand
  ◆ Instance Variables
    ◇ Number: inCount
    ◇ CStackCommand: myCommands[*]
  ◆ Functions
    ◆ Function: Add
      ◆ Parameters
```

```
    ◇ CStackCommand: pCommand
  ◆ Actions
    ◇ Set myCommands[inCount] = pCommand.Clone()
    ◇ Set inCount = inCount + 1
◆ Function: Execute
  ◆ Local variables
    ◇ Number: n
  ◆ Actions
    ◆ While n < inCount
      ◇ Call myCommands[n].Execute()
      ◇ Set n = n + 1
```

Once we have the Command pattern in place, we can do even more. Want to be able to work off-line? Put all commands in an array without executing them and run through it and execute every command when you go on-line? Need a command line interface that supports the same operations as your UI? Write a parser that creates the appropriate commands and executes them. User-defined macros? Add the ability to save and restore CStackMacro objects from a file.

I'll be watching you: The Observer pattern

Let's return to the UI. Two things remain to be solved. First, I want the buttons to be enabled/disabled depending on the state of the stack. If the stack is empty, the Pop and Peek buttons should be disabled; if the stack is full, the Push button should be disabled. Second, the list box that shows the content of the stack needs to be updated, which means that the list box has to be updated as soon as a Pop or Push occurs. This could be handled under SAM_Click of the Push and Pop push buttons; but as soon as we add the possibility of triggering a Pop or Push in any other way, the code to update the list box would have to be included there as well. What we would like to have is some kind of notification from the stack to the list box—and any other object interested—as soon as a change takes place in the stack, no matter how this change came about. The same principle can be applied to the buttons; when the stack changes, it will notify the buttons, and they'll decide whether to enable/disable themselves.

A general way to handle situations such as this one is described by the Observer patterns shown in **Figure 4**. A class that needs to inform other classes when changes occur is derived from the class CObservable. This abstract base class holds an array of objects that want to get notified when the object changes. It also provides functions for clients to add/remove themselves as subscribers to this notification. When the Notify function is called on the Observable, it will call the Notification function on every client currently registered as a subscriber. For this to work, the Observable (and the compiler) must be sure that every object registered as a subscriber really has such the Notification function. Therefore, the clients must all be derived from a common base class, CObserver, which implements the function Notification, which makes a late-bound call to DoNotification. **Listing 20** shows the CObservable class.

Look especially at the __Find function used both by Add and Remove to determine if the observer object is

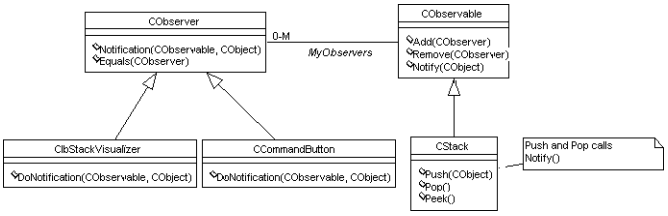

Figure 4. Observer patterns.

already a subscriber. Every object in the array myObservers is compared to the parameter to see if they're equal. This could have been achieved through the use of the standard operator = since CTD 2000 now supports comparing UDVs. This operator determines whether the expressions on the left and right sides of the operator are referring to the same object. Unfortunately, this isn't going to work for us, as we'll see in a minute, since we'll have physically different objects (if such things exist in software) that we want to view as being "the same." Therefore, we introduce the concept that an observer should support comparing itself with another observer through the Equals function.

Listing 20. The CObservable class.

```
◆ Functional Class: CObservable
  ◆ Instance Variables
    ◇ CObserver: myObservers[*]
    ◇ Number: inNextFree
  ◆ Functions
    ◆ Function: Add
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObserver: pObserver
      ◆ Actions
        ◆ If __Find(pObserver) = -1
          ◇ Call __Insert(pObserver)
          ◇ Return TRUE
        ◆ Else
          ◇ Return FALSE
    ◆ Function: Notify
      ◆ Parameters
        ◇ CObject: pObj
      ◆ Local variables
        ◇ Number: n
      ◆ Actions
        ◆ While n < inNextFree
          ◆ If myObservers[n] != OBJ_Null
            ◇ Call myObservers[n].Notification(this,pObj)
          ◇ Set n = n + 1
    ◆ Function: Remove
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObserver: pObserver
      ◆ Local variables
        ◇ Number: n
      ◆ Actions
        ◇ Set n = __Find(pObserver)
        ◆ If n = -1
          ◇ Return FALSE
        ◆ Else
          ◇ Set myObservers[n] =OBJ_Null
          ◇ Return TRUE
    ◆ Function: __Find
      ◆ Parameters
        ◇ CObserver: pObserver
      ◆ Local variables
        ◇ Number: n
      ◆ Actions
        ◇! Check if pObserver is already
          ! subscribing to this object.
```

```
    ◆ While n < inNextFree
      ◆ If myObservers[n] .Equals(pObserver)
        ◇! Found the observer
        ◇ Return n
      ◆ Else
        ◇! Check next slot
        ◇ Set n = n + 1
    ◇ Return -1
◆ Function: __Insert
  ◆ Parameters
    ◇ CObserver: pObserver
  ◆ Local variables
    ◇ Number: n
    ◇ Boolean: bFound
  ◆ Actions
    ◆ While n < inNextFree
      ◆ If myObservers[n] = OBJ_Null
        ◇ Set bFound = TRUE
        ◇ Break
      ◆ Else
        ◇ Set n = n + 1
    ◇ Set myObservers[n] = pObserver
    ◆ If bFound = FALSE
      ◇ Set inNextFree = inNextFree + 1
```

The CObserver class in **Listing 21** is simple; it just implements Notification and Equals, both of which make late-bound calls to the subclass.

Listing 21. The CObserver class.

```
◆ Functional Class: CObserver
  ◆ Functions
    ◆ Function: Notification
      ◆ Parameters
        ◇ CObservable: theSubject
        ◇ CObject: pObj
      ◆ Actions
        ◇ Call ..DoNotification(theSubject,pObj)

    ◆ Function: DoNotification
      ◆ Parameters
        ◇ CObservable: theSubject
        ◇ CObject: pObj
      ◆ Actions
        ◇ Call SalMessageBox("Not implemented",
            "Error", MB_Ok)

    ◆ Function: Equals
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObserver: pObserver
      ◆ Actions
        ◇ Return ..DoEquals(pObserver)

    ◆ Function: DoEquals
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObserver: pObserver
      ◆ Actions
        ◇ Return this = pObserver
```

Listing 22 shows the code for the list box that shows the contents of the stack. It's derived from CObserver and thus must implement DoNotification. The list box must know two things when the notification occurs: What stack just changed? Even though every observer in this example observes just one observable, there's nothing that says it couldn't observe several others, possibly of different types. The stack is passed as the theSubject parameter. Take a look at the Notify function in CObservable and you'll see that the first parameter passed to Notify is the sender itself (using the "this:" reference), in other words, the stack.

The list box must also know what happened to the stack. Was it a pop or a push? This is solved by the second parameter. In this case it's just a number indication the type of event.

Notice that the first parameter is always a reference to the CObservable object making the notification. The second one is just any generic object, and it's up to the observable and its observers to agree on what object to pass and its meaning.

Listing 22. A list box that shows the contents of a stack.

```
◆ List Box Class: ClbStackVisualizer
  ◆ Derived From
    ◇ Class: CObserver
  ◆ Functions
    ◆ Function: DoNotification
      ◆ Parameters
        ◇ CObservable: theSubject
        ◇ CObject: pObj
      ◆ Local variables
        ◇ CStack: aStack
        ◇ CNumber: theEvent
        ◇ CNumber: theElement
      ◆ Actions
        ◆ If SalObjGetType(theSubject) = "CStack"
          ◇ Set aStack = theSubject
          ◆ If SalObjGetType(pObj) = "CNumber"
            ◇ Set theEvent = pObj
            ◆ If theEvent.GetValue() = STACKEVENT_Push
              ◇ Set theElement = aStack.Peek()
              ◆ CallSalListInsert(hWndItem, 0,
                  theElement.GetAsStr())
                ◇ Call SalMessageBox("Error","",0)
            ◆ If theEvent.GetValue() = STACKEVENT_Pop
              ◇ Call SalListDelete( hWndItem, 0 )
```

Listing 23 shows the additions needed to make CCommandButton into an observer and, in that way, have the buttons disable/enable themselves according to the state of the stack. Every time a push or pop is made to the stack, the buttons subscribing to the stack are notified. When the notification arrives, the button needs to determine the state of the stack and enable or disable itself. This could be achieved by having the button interrogate the stack about its state, comparing the capacity with the number of elements to decide if the stack is full, or something similar. We'll try another strategy and change things around so that the button just asks the stack whether the command that the button holds is legal to perform. This is done by a call to the IsCommandValid function. The button doesn't know what rules determine whether it should be enabled or disabled. In this way, if the rules about what commands are legal for a stack in a certain state change, we only have to modify code for the IsCommandValid function in the stack, not any code in the clients of the stack, such as the buttons. Notice that once again the Command pattern gives us a helping hand. The stack knows nothing about buttons; the buttons know nothing about the rules for when a certain command is legal; but both of them know something about commands. **Listing 24** shows that once the IsValidCommand function is implemented, we can use it to enable and disable a menu item and possibly other command triggers as well.

Since different commands are legal in different states, the call to IsCommandValid on the stack will be

forwarded to IsCommandValid in the current state object of the stack, just as with Pop, Push and Peek. This function then makes a late-bound call to DoIsCommandValid that will end up in the correct state subclass (current state of the stack). As an example, **Listing 25** shows the implementation of DoIsCommandValid in CFullStack.

Listing 23. Making the command button into an observer.

```
◆ Pushbutton Class: CCommandButton
  ◆ Derived From
    ◇ Class: CObserver
  ◆ Functions
    ◆ Function: DoNotification
      ◆ Parameters
        ◇ CObservable: theSubject
        ◇ CObject: pObj
      ◆ Local variables
        ◇ CStack: theStack
      ◆ Actions
        ◆ If SalObjGetType(theSubject) = "CStack"
          ◇ Set theStack = theSubject
          ◆ If theStack.IsCommandValid(MyCommand)
            ◇ Call SalEnableWindow(hWndItem)
          ◆ Else
            ◇ Call SalDisableWindow(hWndItem)
```

Listing 24. A menu item using command objects.

```
◆ Menu Item: Pop
  ◆ Menu Settings
    ◇ Enabled when: aPopCommand.GetTarget().
      IsCommandValid(aPopCommand)
  ◆ Menu Actions
    ◇ Call aPopCommand.Execute()
```

Listing 25. DoIsCommandValid in CFullStack state class.

```
◆ Functional Class: CFullStack
  ◆ Function: DoIsCommandValid
    ◆ Returns
      ◇ Boolean:
    ◆ Parameters
      ◇ CStackCommand: pObj
    ◆ Actions
      ◆ Return SalObjGetType(pObj) = "PopCommand" OR
               SalObjGetType(pObj) = "PeekCommand"
```

The adapter

This CObservable code suffers from a major problem: If a window class is derived from CObserver, it still can't be passed as a parameter to the Add or Remove functions, even if they should be accepted as any subclass to CObserver would. The problem here is, of course, that window handles and UDV references are two completely different things. To get around this annoying limitation, we introduce the CAdapter class. This class is derived from CObserver and holds a reference window handle. The call to DoNotification from the CObservable class will be forwarded the real observer, the window handle, by the adapter. When a window object needs to register itself as an observer of the stack, it first creates an adapter object giving itself as parameter and the register the adapter at the stack. **Listing 26** shows the CAdapter class and **Listing 27** shows an example of how a window object would register itself with a stack via an adapter object.

Equality?

Remember the discussion about using the Equals function instead of the = operator in __Find of CObservable? The introduction of CAdapter is the reason for this. If a window object tries to register twice, it will have created two adapter objects, and an ordinary reference comparison will fail to detect that the adapters are actually referring to the same window object. This is solved by having CAdapter implement DoEquals in its own way. The default implementation of DoEquals in the base class CObserver does a plain reference comparison, which will suffice for other observers.

Listing 26. The CAdapter class.

```
◆ Functional Class: CAdapter
  ◆ Derived From
    ◇ Class: CObserver
  ◆ Instance Variables
    ◇ Window Handle: myTarget
  ◆ Functions
    ◆ Function: DoNotification
      ◆ Parameters
        ◇ CObservable: theSubject
        ◇ CObject: pObj
      ◆ Actions
        ◇ Call myTarget.CObserver..
            Notification(theSubject,pObj)
    ◆ Function: SetTarget
      ◆ Parameters
        ◇ Window Handle: phWndTarget
      ◆ Actions
        ◇ Set myTarget = phWndTarget
    ◆ Function: DoEquals
      ◆ Returns
        ◇ Boolean:
      ◆ Parameters
        ◇ CObserver: pObj
      ◆ Local Variables:
        ◇ CAdapter: theAdapter
      ◆ Actions
        ◆ if SalObjGetType(pObj) != SalObjGetType(this)
          ◇ Return FALSE
        ◆ else
          ◇ Set theAdapter = pObj
          ◇ Return pObj.GetTarget() = myTarget
    ◆ Function: GetTarget
      ◆ Returns
        ◇ Window Handle:
      ◆ Actions
        ◇ Return myTarget
```

Listing 27. Using the CAdapter class.

```
Pushbutton Class: CCommandButton
  On SAM_Create
    Call MyCommand.GetTarget().
      Add(CreateAdapter(hWndItem))
```

CP

Johan Alm is a senior consultant with Know IT and has been working with Centura products for the last five years—after leaving the C++ trenches. Other areas of interest include object-oriented development methods, design patterns, frameworks, class libraries, and building guitars. Reach him at johan.alm@knowit.se.