THE CODEBOOK NEWS

January – February 1998

Volume 2, Issue 1

TABLE OF CONTENTS

Editor's Comments

Charles T. Blankenship

The article **Techniques for Creating Prototype Interfaces with Codebook** describes a method of quickly and efficiently developing interfaces, fully operational ones, that users can play with and accept before actual development begins on the production application. One of the large drawbacks I have had to deal with when designing interfaces using RAD techniques is the pervasive belief that the interfaces built during this process are thrown away … fortunately, this is no longer the case. Now, using this technique, interfaces can be quickly created, accepted and their components reused, in their entirety, when development on the actual application begins. This is possible due to the wonderful feature known as inheritance … something VB does ***NOT*** have … yet.

The article, **Rapid Application Development Notes Object**, documents the operation and development of an object that allows you to record the comments made by the user during the interface acceptance review. The previous article described how to quickly build the interface for an application in preparation for the user's accepting it. This article describes a complementary object that allows you (or your user) to record notes about things they like or dislike about your interfaces. These notes are stored in a table, where they cannot be lost (well, odds are good that they won't be), and are available to the developer as a reference when making the changes to the interface that were requested by the users during the walk through.

The article, **Installing Rational Rose and the VFP Modeling Component**, describes how to get Rational Rose, how to install it on your computer, and more importantly, the interface application, available from Microsoft, that makes Rational Rose a two way modeling tool. Since VFP is a part of Visual Studio 97 and since Visual Studio 97 contains a lite version of Rational Rose (they call it Visual Modeler) M$ wrote an application that allows you to reverse engineer your class libraries into a Rational Rose model. And, more importantly, it allows you to convert your Rational Rose class diagrams into VFP class library files (.VCX's). Visual Modeler is available if you own either Visual Basic Version 5.0 Enterprise Edition or Visual Studio 97. The Visual FoxPro Modeling Component, VFPMODEL.APP, is available for download from the VFP owner's area of the Microsoft web site. URLs are provided in the article.

The article, **Understanding The Primary Key Field State**, describes the process that occurs when a record is added to a cursor in the Codebook data environment. It examines Codebook's behavior during an addition of a record and VFP's (not so intuitive) behavior with the default value procedure that can be defined for a field in the Database Container. It also describes the code, originally written by Kevin McNeish of Oakleaf Enterprise Solution Design, Inc., that fixes this problem.

The article, **QuickBooks Date Range Control**, completely documents the operation and development of a very handy interface object present in QuickBooks Version 5.0. This control allows a user to pick from a series of textual date ranges like "Yesterday", "Last Fiscal Quarter", "Last Week", etc. These textual date range requests are then translated into actual date ranges for the user. This prevents them from having to remember the actual number of days in February, or the dates that encompass their last fiscal quarter. This article also illustrates how beneficial a

modeling tool like Rational Rose V4.0.14 can be when communicating object oriented concepts to others, whether the *others* are newsletter subscribers, developers or clients.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

Techniques for Creating Prototype Interfaces with Codebook

Rapid Application Development is a neat way of developing applications. One facet of RAD is the quick development of interfaces that the user can either approve or disapprove … early in the development process. One problem of RAD is that these interfaces, usually, are considered to be throwaways. This is a needless waste of time and software materials … read on for a method of creating prototype interfaces, the components of which are 100 % reusable.

Introduction

Codebook is a true, three-tier development tool. This means that the application objects, developed using Codebook, fit into one of three architectural levels, interface, business rule and data. For this discussion, the interface level is the one of interest. The first goal is to create interfaces, quickly and efficiently, that can be used to get the final approval of the users. The second goal is to accomplish the first goal in such a way that the components can be reused, in their entirety, when production finally begins on the application. Codebook, with only a few modifications, makes this a relatively easy task.

The Requirements

Creating prototype interfaces requires a logical progression from one step to another … here they are:

1) **Basic Requirements Specification** (the details of which will be fleshed out later) – this takes the form of a document, written either by the client or by the developer based upon an interview with the client, that basically outlines the requirements of the proposed system. The purpose of this document is to gather enough of a clue about the proposed system to throw some interfaces together from it. More details of the system will be gathered in the walk through, when the interfaces are complete and the users have an image on the screen to think about.
2) **Data Element Extraction** – this step requires a thorough examination of the requirements specification for the expressed purpose of extracting all possible data elements that will be needed to make the application work for the user.
3) **Data Element Modeling** – in this step, you will classify each data element and identify which framework superclass control is the most appropriate for modeling this data element in the production environment. Create subclasses of these controls and place them in their own application level class library.
4) **Business Object Modeling** – review the requirements specification and determine the business objects that must be modeled to support the application. These business objects are best identified by first looking at the nouns in the requirements specification and identifying the ones that are real world entities that can exist on their own. For example, an employee exists in the real world and is not dependent upon anything else *for* its existence. On the other hand, an employee name requires the existence of an employee for *its* existence. Finally, create each business object and drop the newly modeled data elements onto the most logical business objects.

5) **Form Object Modeling** – next, create the forms that are required to display the business objects to the users
6) **Menu Creation** – create the menu system that activates the form newly created forms.

The Requirement Specification

The purpose of the requirement specification is to get enough information to build enough of an interface to get the user's imaginations flowing. This can be a written document prepared by the user's themselves or a written document prepared by the developer after an initial interview. How you get the spec. is not important. The fact that you *should* get a written description of what the users want their system to do cannot be overemphasized.

The following is the requirement specification for an extremely simple employee application for which the interface must be developed and accepted. This is what this article will use to illustrate this process:

> We need a way of tracking the names, addresses, phone numbers, birth dates, hire dates and emergency contact information for our employees.

A critical note must be injected here. This initial requirement specification does not have to be a 100 page document that defines, to the n^{th} degree, each and every behavior of the system. If you can get this type of document from your users, great, but your chances of this are slim. You are much more likely to get a sketchy list of "I'd like to have... " statements that must be detailed, considerably, before work on the custom system can begin. This process describes how to create an initial interface that you can use to facilitate the "detailing" process. The trick is to create this interface in a period of time that is viewed as trivial (read inexpensive) by the client.

Data Element Extraction

This step requires a detailed examination of the requirements specification for the expressed purpose of creating a list of all the data elements required by the application. Add elements to the list, if you want to, by including additional data items not definitively present in the actual requirement specification, but are implied by the content. For example, the above requirement specification talked about an employee's address. The data elements required to effectively model an address are two address lines, city, state and zip code data elements. Include them in your data element list. Compose this list in such a way that you have enough room to add a second column on the right side. This will be needed for the next step.

After viewing the requirement specification, the following data elements are required to model the employee system:

Data Element	Framework Superclass
Employee First Name	SAVIContainedTextBox
Employee Middle Initial	SAVIContainedTextBox
Employee Last Name	SAVIContainedTextBox
Employee Address Line One	SAVIContainedTextBox
Employee Address Line Two	SAVIContainedTextBox

Employee City	SAVIContainedTextBox
Employee State	SAVIContainedStateComboBox
Employee Zip Code	SAVIContainedZipCodeTextBox
Employee Phone Number	SAVIContainedPhoneNumberTextBox
Employee Birth Date	SAVIContainedDateTextBox
Employee Hire Date	SAVIContainedDateTextBox
Emergency Contact First Name	SAVIContainedTextBox
Emergency Contact Last Name	SAVIContainedTextBox
Emergency Contact Relationship	SAVIContainedTextBox
Emergency Contact Phone Number	SAVIContainedPhoneNumberTextBox

Data Element Modeling

This step requires you to examine the data elements proposed for the new application and identify which framework superclass control is the best one to use to model the data element. In the second column of the list you created in the previous step, write the name of the framework class definition you want to use to model the data element. Notice that the classes named above are SAVI controls. These controls are subclasses of the controls present in cControl.VCX and are used to define the standards for controls created by Software Assets of Virginia, Inc. If you have not created this abstraction level for your own company, you should. It creates a level of abstraction where you can define the characteristics of your company's standards when building applications and removes these changes from the cControls.VCX. If you do this, you can easily replace the framework cControl.VCX, with a new one, and keep your company's standards intact.

Use Inheritance for Maximum Flexibility

Inheritance is one of the most powerful of the new tools available in Visual FoxPro, and one of the most underutilized, unfortunately. I have never got into trouble by subclassing too much, although this is possible. I have frequently had my development hampered by not subclassing enough. The key to accurate subclassing resides in the justification for doing so. If you can describe, accurately, *your* reason for subclassing and you can justify it to *your* satisfaction, then you should subclass without reservation.

Use Inheritance for Application Specialization

The next step in this process is to create a unique list of superclass controls, identified in the previous steps, and create application level subclasses for each one of them. A list of superclass controls that should be subclassed into application level controls follows:

	Framework Control Classes	Application Level Control Classes
1)	SAVIContainedStateComboBox	MRSStateComboBox
2)	SAVIContainedZipCodeTextBox	MRSZipCodeTextBox
3)	SAVIContainedPhoneNumberTextBox	MRSPhoneNumberTextBox
4)	SAVIContainedDateTextBox	MRSDateTextBox
5)	SAVIContainedTextBox	MRSTextBox

Place each new subclass in an appropriately named class library, I use aControl.VCX, and give them a name that is a derivative of the client's name. Place this application level class library in

the LIBS subdirectory of the application's development subdirectory. Examples of the controls created for **M**. **R**osenblatt and **S**on, Inc are displayed in the second column above.

The justification for performing this subclassing is to keep client interface standards out of the framework. In other words, many clients are satisfied with the attributes of the SAVI controls and do not require any additional modifications. But what if, in the future, a change of management comes along and requires the bolding of a control labels. If this application level subclass did not exist, I would have only one of two choices. The first is to change the .FontBold attribute at the framework level of abstraction, directly in the cSAVICCt.VCX. The second choice is to make this change on every control on every business object in the entire application. Neither choice is acceptable. The first choice differentiates the framework superclasses from one client to another. This means that if I ever create an enhancement to the framework that I'd like to give to my clients as a free upgrade, I have to remember which client's had which modifications made to which properties. I would then have to recreate each modification after I replaced the cSAVICCt.VCX in their application. This is immediately unmanageable. If I had taken the time to perform an application level subclass of the controls, this problem could have been prevented. The second choice, modifying each control present on each and every business object, simply takes too long and illustrates that I am not taking advantage of the power of true object oriented programming.

Creating an application layer of abstraction allows me to give my users updates without having to worry about overwriting their application level modifications. It also allows me to make those application level modifications in one place and have the results echoed throughout the application while leaving the framework level classes unchanged. It takes about 5 to 10 minutes to create these subclasses; this is time well spent.

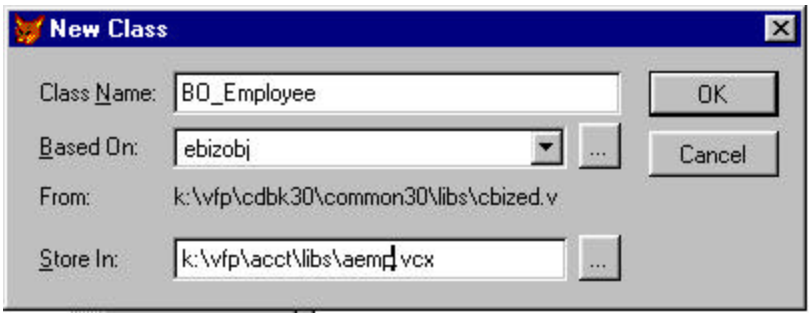
Business Object Modeling

The next step is to examine your requirements specification and try to identify all of the objects that are tangible to the application. When I talk about a tangible object I refer to those objects that can exist on their own. Examples of tangible objects are invoices, customers, employees, clients, tasks, appointments, etc. Example of data elements that are not tangible objects include employee first name and last name, begin time of the appointment, invoice number, et cetera.

Separate Class Libraries for Tangible Objects

Once you have identified each tangible object in the application, create a separate class library to model that object. For example, if the application you are building requires the modeling of customers, employees and invoices, create three application level class libraries to store the components that will be used to model these objects.

The following figure illustrates the information you must provide to define a new Employee business object. To access the New Class dialog, go to the Classes tab of the project manager and press the New button. A New Class dialog, like the one illustrated below, appears. In this situation, a new Employee business object is being modeled. It is based on Ed Leafe's eBizObj and will be stored in a class library named aEmp.VCX. The class library itself is stored in the application's LIBS subdirectory.



Use Inheritance to Model Application Data Elements

The next step is to model each data element identified for the Employee business object. These include first and last name, middle initial, address, city, state, zip code, et cetera. However, instead of creating your application data elements by inheriting from the framework level control classes, cTextBox, cComboBox, etc, you should use the newly created, application level control classes as superclasses. In this example these are the MRS* class definitions contained in the aControl.VCX class library (present in the LIBS subdirectory of the application).

Initially, you only need to provide the most basic information for these data elements. The only properties you must populate are the **`.Caption`** property, for the control's corresponding label, and the **`.Width`** and **`.Height`** of the control itself so it displays the approximate size of the data to be entered. The control does not have to be bound to a **`table.field`** yet, (leave the **`.ControlSource`** property empty) which is fortunate because there are no tables or fields defined at this point. The normalization process has not yet been completed. Modifications made to the Codebook framework by SAVI enable you to instantiate controls and business objects without corresponding **`table.field`** names.

When you are completed with this step, you will have a class library that contains all of the component objects required to model the real world object. The primary benefit of architecting your application in this manner is during the maintenance phase. When another programmer views the classes tab of you project, they will no longer see just an aBizness class library. They will see class libraries whose names (or at least descriptions) approximate the real world objects being modeled by the application, aInvoice, aCustomer, aPurchase, etc. Using this methodology makes the project itself an abstraction of the business you are modeling.

Notice that the above advice departs from the contents of the initial help file, authored by Paul Bienick, developer of much of the Codebook framework. There, he advocated dragging and dropping framework level control class definitions directly onto the business object being modeled and making application level property modification there. This, in retrospect, is not an advisable procedure since it severely limits the reusability of the interface objects.

One of the reasons that interface prototypes are thrown away is that you will not get the interface design right the first time. Guaranteed. The users will let you know exactly how bad your initial design really is (unless you are a Tom Meeks clone) during the walk through. You will be asked to rearrange interface objects, move them to other business objects, etc. You must plan for this or

pay by throwing your previous work away. Unless you have a distinct data element object to work with, you are going to have to do a lot of retyping before modifications of this nature are possible. This additional time makes rapid interface prototype development uneconomical and therefore unacceptable to the client.

This problem can be alleviated by subclassing one more times and creating a distinct object for each data element in the application. With a tangible object with which to work, moving a data element from one BizObj to another is a simple as deleting it from the offending BizObj and dragging and dropping it onto the desired BizObj.

As a side note to those of you who are concerned with the performance impact that this amount of subclassing can have on your applications, rest assured that the impact is very small. I created a control, in the above fashion, but subclassed it *ten* times. I then used the 10th level control abstraction five times on a business object. I then created another business object and used five first level abstractions there. There was no discernable performance difference between the two on a P-133MHz Toshiba Satellite Pro Model 440CDX laptop. Therefore, the performance impact of two levels of abstraction should cause no one any problems, only completely *justified* benefits.

Business Object Modeling

The next step is to place the newly modeled data elements onto their respective business objects. Using the previous example this would entail modifying the Employee business object and dragging and dropping its associated data elements onto it. Arrange them in a visually appealing fashion. Finally, save the newly created business object. Notice that all of the elements required to model an Employee object are contained in the same place, the aEmp.VCX. Any time a maintenance programmer needed to go into your application and maintain an employee object, they only need to concentrate their efforts in one place, the aEmp.VCX.

If you are interested, you can also apply the same subclassing logic to business objects that was applied to interface objects. If you do, you should subclass eBizObj into an application level eBizObj, MRSBizObj in this case, and then inherit from the new application level business object to create all of the additional business objects required to model the business. Since the performance impact is negligible, this additional level of abstraction can come in very handy at times. When you end up using them, it can make you look like you had the forethought, when necessary, to handle modifications with ease. Following on with this logic, you can do the same thing with the form objects as well.

Form Object Modeling

The next to last step in this process is arranging the business objects on the forms. Notice that since SAVI uses a variation of eBizObj, a derivation of cBizObj created by Ed Leafe of Ed Leafe Consulting, that business objects can be placed anywhere on a form that you desire. You can bury them in page frames, you can bury them within one another. Anything that you want to do to accomplish the desired interface effect is possible with eBizObj.

Therefore, subclass your forms, either from application level form subclasses or framework level form classes, whatever you prefer, and drop your business objects onto them in an aesthetically pleasing manner. Save your changes and ready your self for the final step.

Menu Creation

The final step is to hook your forms into the menu system. Once you are done with the menu system, you are done with the rapid application development of the first interface prototype of your new application. Show it to your users and get their acceptance before you go any further in the development process.

Conclusion

It took much more time to write about this process that it actually takes to accomplish it. In the test application, the amount of time it took to accomplish the necessary tasks are as follows:

Perform the application level subclassing	Five minutes
Create the data element interface objects	One minute a piece, approx. 15 min.
Create and populate the BizObj with data elements	Five minutes
Form creation	Five minutes
Menu Creation	Five minutes

… for a total of 35 minutes.

Some of you may say that this period of time is too long. If these minutes were *wasted* on an interface that was going to be thrown away, I might have to agree. However, each and every piece created during this process will be reused, in its entirety, when production begins on the application. Therefore, this time was *not* wasted. You must perform each of the steps presented above sooner or later. In this case, the sooner the better since this effort goes into a product that you and your clients can use to agree on the interface design *before the first line of business logic is written*. More importantly, using the enhancement discussed in a following article, **Rapid Application Development Notes Object**, this interface can be used to collect development notes during the initial walk through. Being able to take detailed development notes and get the initial interface accepted *before* any business code is written is invaluable to the success of many application development efforts.

Notice that several enhancements had to be made to Codebook to get it to behave properly during this process. These enhancements were not discussed in this article for reasons of clarity. If you are interested in receiving a free copy of the SAVI version of Codebook, simply fill out the application form at the end of this article, follow the instructions and send it to SAVI. You will receive a free, unlimited copy of the framework SAVI uses to develop applications for its clients.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

Rapid Application Development Notes Object

Once the prototype interface is completed, it is time to do one of many walkthroughs with the user. It is very important to capture every concern that they may have with the interface. This object, included in SAVI's version of Ed Leafe's eBizEd.VCX, makes taking notes during these walkthroughs a breeze.

Introduction

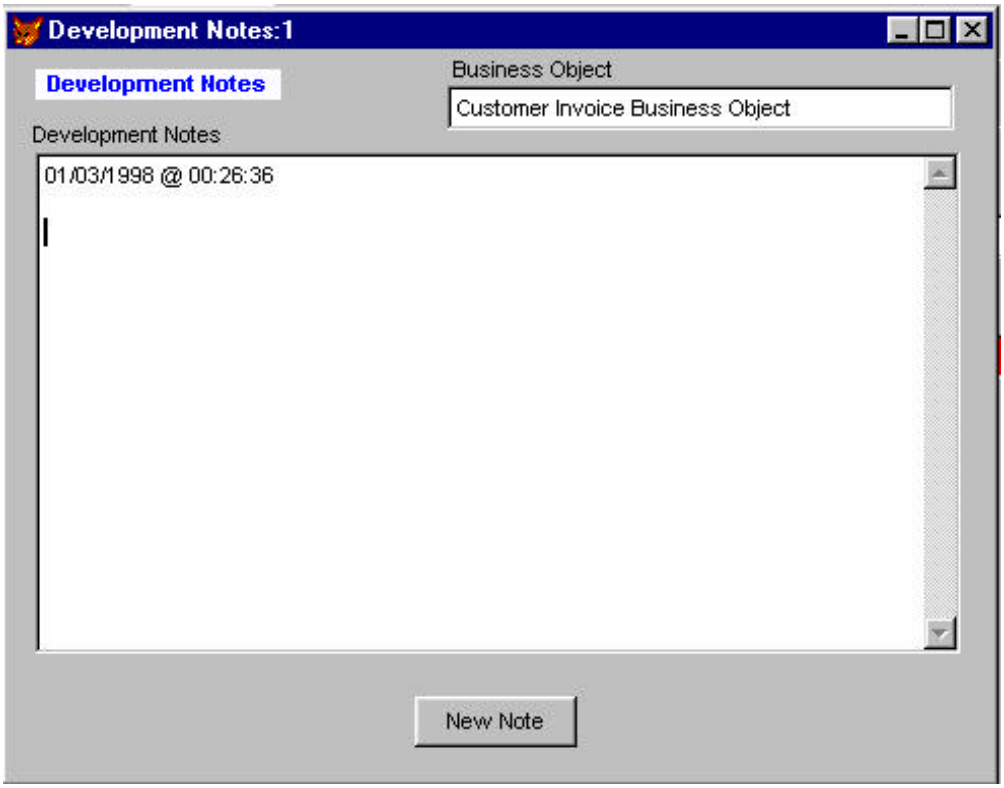
The software interface is one of the most critical parts of a successful application. There are many right ways and even more wrong ways to create them. Your users could interpret what seems like a perfectly good interface to you, as a less than optimum. One tried and true method of creating a successful interface is to design a prototype and let the users accept it before you write one single line of code, or design one entity relationship diagram. However, you must prepare yourself to *record* and implement the changes the users may desire … *after* viewing your prototype.

You have several choices. One is to write all of their requests down on a piece of paper. Another is to Alt+Tab back and forth between Word and your application. Neither of these options was as integrated as I wanted this process to be. So I decided to write a short enhancement to Codebook that tightly integrated the note taking process with the newly created interface.

Using the RAD Notes Object in Development

The use of this object, from the developers point of view, could not be simpler. If you are using SAVI Codebook, simply follow these steps:

1. Populate the .**`cDescriptiveName`** property of your business object with a name the user is comfortable seeing. For example, if you are designing the interface for an invoice header business object, populate this property with **`Customer Invoice Business Object`**. You may wish to leave the "Business Object" phrase off of the descriptive name value since your users may not know what a "business object" is.
2. Create a text file, named DEVNOTES.TXT in the application's main directory. Do this by typing MODI FILE DEVNOTES.TXT, type some gibberish in the file, and save it. The content of the file is not important, only its presence is required to ensure the Development Notes object is available on all business objects in your prototype interface.
3. When the business object appears on its form, it will have a white label with blue lettering that says "**Development Notes**". Click on that and the following form appears. Notice that the descriptive name you gave to the business object appears at the top of the Development Notes form. The record that stores the development notes for this business object is automatically created if it does not already exist. It is automatically displayed if the record already exists from a previous note taking session.



This form consists of a simple business object, which contains a text box, an edit region and a command button. The "Business Object" text box stores the unique identifier for the notes you will take about this business object. The "New Notes" button allows you to create a new date time stamp for a new series of notes by a simple mouse click. This comes in handy when taking additional notes on different days … it shows a history of note taking interaction with the user. Finally, the development notes themselves are typed into the "Development Notes" edit region.

Since this interface was designed around the Codebook standard, you can do anything to these notes that you want, Create new ones, Read, Update or Delete existing notes. Previously, I informed you that a note for a business object is created automatically if it doesn't already exist in the table, but don't think you are limited to creating notes only for corresponding business objects. You can press the File | New button and create a new note that defines your To Do list for this prototype review session as well.

The nice thing about using the Codebook interface is that multiple forms for multiple business objects can be opened at once. You can also use the navigation buttons to navigate between the different notes created for the application. Using this method, you can capture all of the criticisms and suggestions that your users may want to make about the interface you have proposed. Most importantly, these notes can never be accidentally thrown away, or lost. They are always with the application.

Conclusion

The most amazing thing I found out about this process was the massive amount of detail I was able to extract out of the users once they were in front of a "working" application. I have sat in design meetings where white boards were used to "create" interfaces, and I have recently had the chance to use this method. The two cannot be compared. *Something* happened to the users that sat in front of the "working" application that did not occur in the whiteboard design meeting. It was as if just setting in front of the application's interface, viewing an approximation of what they will see in the final application, opened my clients up like flowers. I built a prototype interface, using the previously documented method, from a one page, type written document. Once I was completed with the interview, I had 15 full pages of type written information (in Word) about the same application (it was a small system).

Before a single line of code was written, I had a detailed requirements document that specified exactly what the users wanted. I also had their approval on the first iteration of the interface. Probably one of the greatest benefits of this approach was to make the user's feel like they played a significant part in the design of their own system. They were excited to see the finished product … they became champions of their own software. This made my job much easier.

I have said this before and I'll say it again here. *Programming*, in itself, is **NOT** the most difficult work of developing applications … getting the users to tell you **WHAT** they *really* want is one of the most difficult tasks we are ever faced with. Hopefully, this enhancement will ease the burden of discovering what that ***what*** really is … at a minimal cost to both you and your client. It will, hopefully, cast the new system in a very favorable light as early in the development process as possible.

If you are interested in receiving the software for this enhancement to Codebook, simply fill out the application form at the end of this issue and mail it to SAVI. You will receive download instructions, complete with passwords, via eMail.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

Obtaining Rational Rose and the VFP Modeling Component

As applications become more and more complex, application developers must begin to use modeling tools to create and communicate their design ideas to clients as well as one another. One excellent package on the market today is Rational Rose, a visual modeler developed by Rational Software Corporation. What you may not know is that Rational Rose and Microsoft Visual FoxPro talk quite nicely to one another.

Introduction

When I began working with FoxPro, I had the good fortune to have an excellent mentor. One of his areas of focus was the use of CASE tools to help engineer software. He taught me quite a bit about how to use them effectively. I also had the good fortune to talk to Whil Hentzen at a Virginia Beach, VA developers conference. When I asked him if he used CASE tools, he said he did not. His reason, at the time, was justifiable. He pointed out that much of the energy put into designing an application using a CASE tool could not be readily leveraged to help create the components of an application. He was right. A good deal of time and effort was required to extract the design out of the CASE tool to create the application. This drawback, luckily, is beginning to fade away.

A Benefit of VFP's Inclusion in Microsoft's Visual Studio

Everyone is beginning to understand the benefit of CASE tools to software development. To this end, Microsoft has included a modeling tool in their Visual Studio. They call it Visual Modeler (VM). It just so happens that VM is really a licensed subset of, you guessed it, Rational Rose Modeler. Luckily, since VFP and Visual Modeler both exist within Visual Studio, Microsoft has written an interface (an .APP really) that allows VFP to talk to Visual Modeler and Visual Modeler to talk to VFP. The means that you can import your existing VFP class libraries into Rational Rose. More importantly, it means you can export your class designs from Rational Rose into VFP, i.e. create real .VCX files from visual designs. Finally, the barriers that previously prevented application developers from using CASE tools are beginning to disintegrate. Read on for instructions for downloading and installing these two cool tools.

Obtaining Rational Rose from Rational Software

You have several options here.

1. You can download it from their web site, http://www.rational.com/downloads/. Scroll down the page until you see the heading Visual Modeling with Rational Rose. Click on the hyper link that says Rational Rose 4.0 Demo. This will take you to the download page. Click on the green bullseye.
2. Call them and they will mail you the same software on a CD-ROM. The number for this is (800) 728-1212
3. Buy it for $1,600.00 plus shipping and handling
4. Buy Visual Basic Enterprise Edition, Version 5.0 or Visual Studio (these have scaled down versions of Rational Rose Modeler)

Obtaining the VFP Link from Microsoft

First, do not call Microsoft and ask about this product. Even their technical sales folks didn't have the slightest clue. As a matter of fact, the only way I found out about it was from a Rational Software salesman. The following is an excerpt from Rational's web site, the URL for this page is http://www.rational.com/products/rose/roselinkprod2.html.

Rose Visual FoxPro Link from Microsoft Corporation Microsoft develops Visual Modeler, a modeling tool for representing large-scale component-based applications, the result of a development collaboration with Rational.

In order to get this software (a wizard actually), you must be a registered owner of VFP and have the capability to get onto the VFP Owner's area. The URL that begins your journey is as follows: http://www.microsoft.com/vfoxpro/owners/. The page looks like this ...

Additions for the week of September 29, 1997

Updated Modeling Wizards

Updated Version of Microsoft Visual FoxPro Modeling Wizards **< < - - You want this one**
Visual modeling makes it possible for developers to manage the complexity of component-based development with an architecture-driven approach. Developers using Microsoft Visual FoxPro 5.0 will further benefit from the Visual Modeler technology with the addition of Visual Modeler Connection Wizards. This release of the Reverse Engineering and Code Generation Wizards is for compatibility with the new Microsoft Visual Modeler 1.1.

Additions for the week of April 18, 1997

Modeling Wizards

Now Available: Full Release Version of Visual FoxPro Modeling Wizards
Visual modeling makes it possible for developers to manage the complexity of component-based development with an architecture-driven approach. Developers using Microsoft Visual FoxPro 5.0 will further benefit from the Visual Modeler technology with the addition of Visual Modeler Connection Wizards. Full release versions of the Reverse Engineering Wizard and the Code Generation Wizard are now available in the Owners' Area.

Once you click on the above hyperlink, you will be transported to the place where you can access the owner's area. Chose to enter the Visual FoxPro site and look for the following text.

Updated Version of Microsoft Visual FoxPro Modeling Wizards

Visual modeling makes it possible for developers to manage the complexity of component-based development with an architecture-driven approach. Developers using Microsoft Visual FoxPro 5.0 will further benefit from the Visual Modeler technology with the addition of Visual Modeler Connection Wizards. This release of the Reverse Engineering and Code Generation Wizards is for compatibility with the new Microsoft Visual Modeler 1.1.

Read the Visual Modeler Press Release, 3/18/97
Download the updated self-extracting Wizards (147k) **<<- You want this one**

Once you obtain the software from these two vendors, follow the installation instructions. The installation of both products is standard and straight forward. No additional aid should be needed. A subsequent article describes how to reverse engineer VFP class libraries as well as how to generate class libraries from your class diagrams. I have discovered a view tricks that can make your life much easier.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

Understanding the Primary Key Field State Situation

Codebook, when it adds a record to the data environment, changes the Field State of the Primary Key from Appended Modified to Appended, Not Modified. This can create a problem where the primary keys are used twice, once when the cID is pre-fetched in the OnNew() method and again when the record is saved in the Save() method. This article explains this behavior and the fix, created by Oak Leaf Enterprises Solution Design, Inc.

Introduction

In order to understand this process, you must know how Codebook creates a new record in the data environment as well as what happens when that new record is saved and what can happen in between.

Adding a New Record

Whenever a user decides to add a new record, a New() message is sent to the form. From there, the New() message is sent to the business object. Once the New() message reaches the business object, the following code is executed.

Listing One: eBizObj::OnNew()

```
LOCAL lnOldWA , ;
      lnRetVal

lnOldWA = SELECT()
SELECT ( THIS.cAlias )

lnRetVal = FILE_CANCEL

IF THIS.AllowNew()
   *==============================
   lnRetVal = THIS.oBehavior.New()
   *==============================
   IF lnRetVal = FILE_OK
      THIS.OnNew()
      IF THIS.lSetFocusOnNew
         THIS.SetFocusToFirst()
      ENDIF
   ENDIF
ENDIF

IF lnRetVal = FILE_OK
       THIS.RequeryAllChildren()
ENDIF


IF THIS.lNewChildOnNew
```

```
   THIS.refBizObj.New()
ENDIF

SELECT ( lnOldWA )

RETURN lnRetVal
```

This code, first checks to see if the user is allowed to add a new record. If so, the business object's data environment object is told to add a new record to the data environment. If the new record was successfully added, the business object's OnNew() method is called and post add processing is initiated.

On New Processing

The subclassed business object's OnNew() method is the place you should put all of the code that performs actions on a newly added record. One of the default functions is the pre-Fetching of a primary key value from either the .DBC, if a local view is being added, or the backend database if a remote view is being added. In this case, once a new record is added, the default value of its primary key field must be generated and placed in the cID field. The other default function, is the capability of automatically populating the record's foreign key value with the primary key value of the parent, this is the subject of another article. The code for the eBizObj::OnNew() method follows:

Listing Two: eBizObj::OnNew() method

```
THIS.FetchPrimaryKey()
IF THIS.lAutoPopulateForeignKey
   THIS.PopulateForeignKey()
ENDIF
```

Notice the benefits provided by the technique of programming in the problem domain. Programming in this fashion results in a high level of self documentation within the code. It takes about two seconds to answer the question "What happens when a new record is added to the data environment?" Two things, the primary key is fetched from the database and the foreign key is populated if the programmer desires this to occur.

Fetching the Primary Key

The following is a listing from the eBizObj::FetchPrimaryKey() method. It, too, has been rewritten using the concepts of programming in the problem domain. See how long it takes you to figure out what is being accomplished.

Listing Three: eBizObj::FetchPrimaryKey() method

```
IF lnCursorType <> DB_SRCTABLE

   lcBaseTable    = ""
   lcPrimaryKey   = ""
   lcDefaultValue = ""

   THIS.GetDefaultPrimaryKeyExpression( lnCursorType  , ;
                                        @lcBaseTable  , ;
                                        @lcPrimaryKey , ;
                                        @lcDefaultValue )

   llKeyGenerated = THIS.ReplacePrimaryKeyValue( lcPrimaryKey , ;
                                                 lcDefaultValue)
   IF llKeyGenerated
      THIS.SetFieldState( lcPrimarykey , ;
                          FLDSTATE_APPEND_NOTEDITED )
   ENDIF

ENDIF  &&-- lnCursorType <> DB_SRCTABLE
```

Here is an explanation of the code:

1. If the cursor in the current work area is not a table, i.e. it is a local or remote view.
2. Get the Default Primary Key Expression and let me know what the name of the base table, primary key and the expression that generates the default value is.
3. Replace the Primary Key Value with the Default Value.
4. Finally, if the key was successfully generated, Set the Field State of the primary key to be appended and not edited.

Resetting the Primary Key Field State

Setting the field state of the primary key is the area of interest. The code for this method is listed below:

Listing Four: eBizObj::SetFieldState()

```
LPARAMETERS tcFieldName , ;
            tnFieldStateValue

=SETFLDSTATE( tcFieldName, tnFieldStateValue )
```

The only thing this code does is change the field state of the specified field to the specified value. In this situation, the specified field is cID and the specified value is 3.

As soon as the new record is added, if you suspend program execution and examine the field states by issuing the command GETFLDSTATE(-1), you will see that all of the fields have a 3, which indicates that they are part of a newly appended record and that no fields in that record have been edited. As soon as the new value for cID is generated and placed in the cID field, the field state for that field changes from a 3, appended not edited, to a 4, appended edited. The code

present in the SetFieldState() method then changes the field state from appended edited to appended not edited immediately after generating and placing the new ID value in the cID field. The first question that leaps to mind is why?

Why Change the cID Field State Back to Appended Not Edited

The reason lies in eBizObj's IsChanged() method. The following is a partial listing of that code.

Listing Five: Partial Listing of eBizObj::IsChanged()

```
IF CURSORGETPROP( "BUFFERING" ) = DB_BUFOPTTABLE
   llIsChanged = ( GETNEXTMODIFIED( 0 ) # 0 )
ELSE
   IF !ISNULL( GETFLDSTATE( -1 ) )
      llIsChanged =  ( "2" $ GETFLDSTATE( -1 ) OR "4" $ GETFLDSTATE( -1 ) )
   ENDIF
ENDIF
```

Notice the highlighted code. In order to determine if a record in the current work area has been changed or not, the value returned from the GETFLDSTATE(-1) function is examined for the presence of either 2's or 4's. As previously stated, a 4 indicates that one of the fields in the appended record has been edited. As soon as a record completes the add process, the only changes the newly added record has undergone is the addition of the primary key. Not exactly a user level change. It is for this reason that the field state of the primary key is changed back to appended not edited once the value for the ID is placed in the cID field of the new record. This ensures that the IsChanged() method only returns a .T. if the *user* makes a change to the record once the record is added. It should return a .F. if the only change made to the record is the population of the cID field by the framework itself. To find out why this is necessary, search eBizObj and eBizObjForm for all calls to IsChanged(). My search turned up the following:

1. eBizObjForm::QueryUnload()
2. eBizObjForm::First()
3. eBizObjForm::Prior()
4. eBizObjForm::Next()
5. eBizObjForm::Last()
6. eBizObjForm::Save()
7. eBizObjForm::New()
8. eBizObjForm::PerformPreRequeryConditioning()

Therefore, if the field state of the primary key was not changed back to a appended not edited immediately, any time the user tried to close the form, move the record pointer, save the record or add a new record, the framework would detect a change and perform accordingly. *Accordingly*, is usually a call to the AskToSave() method of the form. This is true in all of the above methods except the Save() method. If the Save method is called and there are no changes to save, the save is canceled … makes sense.

The Ramifications of Setting the Primary Key Field State to Appended Not Edited

Now that you know why the framework changes the primary key field state from appended edited to appended not edited (prevent unwanted messaging, saving, etc.), it is time to find out the

potential problems that exist when the user makes their changes and executes the Save() method. If the state of the primary key is left as appended not edited, VFP generates a second value for cID when the record is Saved().

In order to understand why this happens, you must realize what causes VFP to trigger the default field value processing. When determining whether or not to generate the default value for a field in a new record, VFP *only* looks at the current setting of the field state. It ignores the actual *content* of the field. **The default value is only generated if the field state is 3, appended not edited. It is not generated if the field state is 4, appended edited. Therefore, changing the field state back to appended edited, before the Save() method executes, prevents FoxPro from overwriting already provided values with default values, i.e. using up a second cID value. Default values should only be generated if the field has not been edited by the user.**

A Short Recap

When a new record is added to the data environment in Codebook, the field state for the cID field is initially 3, appended not edited. Then, when the primary key value is placed in the cID field, the field state changes to 4, appended edited. Next, the Framework changes this value back to 3, the user makes their changes and executes the Save() method to save them. When the TABLEUPDATE() function is executed from within the Save() method, VFP examines the field state of the cID field, sees that it is 3, appended not edited, and says "Hey, trigger the default value expression for the field". It ignores the fact that a unique identifier is already there. The result is that the NewID() function triggers a second time and in the process, uses up the second unique identifier for this record.

To fix this problem, the only thing that must be accomplished is to change the field state for the cID field from 3 back to 4. Then, when the TABLEUPDATE() function is triggered, VFP says "Hey, a value has already been provided for the cID field, no need to trigger the default value expression." The NewID() function is not triggered a second time and the problem is solved.

Changing Primary Key Field State Back to Appended Edited

To accomplish this, some pre-save processing must be executed that changes the field state for the primary key from appended not edited, 3, to appended edited, 4. The hook for this processing is located in the cDataBehavior::Save() method.

The general process for this method is to first check if the Save() is being executed against a local or remote view. Views are the only type of cursor for which this processing is required. If the Save() is being processed against a view, the next check is to determine if the cursor is table buffered. If the cursor is table buffered, scan through each record and check if it is a newly added record (this is the purpose of the IsAddingOriginal() function). If it is a newly added record, change the state of the primary key field back to appended edited. This prevents the NewID from firing the second time when the TABLEUPDATE() function is executed.

Listing Six: Complete listing of cDataBehavior::Save() method

```
LPARAMETERS tlAllRows, tlForce
```

```
LOCAL lnRetVal, ;
      laError[AERROR_NUMCOLUMNS], ;
      loSelect, ;
      lnOldRecNo

*-----------------------------------------------------
*           LOCAL/PRIVATE VARIABLE DESCRIPTIONS
*-----------------------------------------------------
* lnRetVal - file status ... see FRAMEWRK.H Constants to
*            identify file status.
* laError[] - stores the information returned from AERROR[]
*             Column 1 stores the error number.
* loSelect - stores the currently selected work area
*-----------------------------------------------------

*-----------------------------------------------------
*-- We save and restore the work area since VFP
*-- will not restore the work area if a validation rule
*-- fails during a TABLEUPDATE().
*-----------------------------------------------------
loSelect = CREATEOBJECT("CSelect")

*-----------------------------------------------------
*-- Mark primary key as being updated if using a view.
*-- This is necessary if we're pre-fetching primary key
*-- values.
*-----------------------------------------------------

IF CURSORGETPROP("SOURCETYPE") <> DB_SRCTABLE
       *-------------------------------------------
       *-- If cursor is table buffered, scan all
       *-- records, and set the state of the
       *-- primary key field to "Edited"
       *-------------------------------------------------
       *-- Added the check for pessimistic table buffering
       *-- 10/14/1997 - 12:42a
       *-------------------------------------------------
       IF CURSORGETPROP('BUFFERING') = DB_BUFOPTTABLE OR ;
          CURSORGETPROP('BUFFERING') = DB_BUFLOCKTABLE

       *-------------------------------------------
       *-- MODIFIED - 10/22/1997 - 1:10
       *-- Added code that repositions the RP to its
       *-- original position.
       *-------------------------------------------
       lnOldRecNo = RECNO()

       SCAN
          IF IsAddingOriginal(ALIAS())
             This.SetPrimaryKeyFieldState( FLDSTATE_APPEND_EDITED )
          ENDIF
       ENDSCAN
```

```
      *----------------------------
      *-- Reposition record pointer
      *----------------------------
      DO CASE
      CASE lnOldRecNo = 0
      CASE lnOldRecNo > RECCOUNT()
      OTHERWISE
         GO lnOldRecNo
      ENDCASE

      ELSE
            IF IsAddingOriginal(ALIAS())
                  THIS.SetPrimaryKeyFieldState( FLDSTATE_APPEND_EDITED )
            ENDIF
      ENDIF
ENDIF
*--}
```

```
*--------------------------
*-- Try to update the cursor
*--------------------------
IF TABLEUPDATE( tlAllRows , tlForce )
  lnRetVal = FILE_OK
ELSE
  *----------------------------------------
  *-- Return error number of what went wrong
  *----------------------------------------
  =AERROR(laError)
  lnRetVal = laError[AERROR_ERROR]
ENDIF
RETURN lnRetVal
```

The code for SetPrimaryKeyFieldState() follows, this was written by Kevin McNeish of Oakleaf Enterprises Solution Design, Inc. and effectively fixed this little problem.

Listing Seven: Complete Listing of cDataBehavior::SetPrimaryKeyFieldState()

```
LPARAMETERS tnFieldState
LOCAL lcPrimaryKey          , ;
      lnPrimaryKeyFieldState , ;
      lnOKToSet
*-----------------------------------------------------------------
*--- Get the list of key fields for the currently selected cursor
*-----------------------------------------------------------------
lcPrimaryKey = CURSORGETPROP("KEYFIELDLIST")
llOKToSet    = .F.
*-----------------------------------------------------------------
*--- Ignore all other keys in list except the first 09/24/96 - KJM
*-----------------------------------------------------------------
IF ',' $ lcPrimaryKey
       lcPrimaryKey = SUBSTR(lcPrimaryKey,1,AT(',',lcPrimaryKey)-1)
ENDIF
*-------------------------------------------------------
*-- Determine what the field state is for the primary key
*-------------------------------------------------------
lnPrimaryKeyFieldState = GETFLDSTATE( lcPrimaryKey )
*-----------------------------------------------------------------
*-- Only update the field state of the primary key if
*-- the state of the record (appended (3 or 4) / existing (1 or 2)
*-- matches the desired field state.  In other words, don't
*-- allow the programmer to attempt to set the field state of
*-- an existing field from one of the existing to one of the
*-- appended states
*-- CTB – 10/20/97
*-----------------------------------------------------------------
DO CASE
CASE ( tnFieldState = 1 OR tnFieldState = 2 ) AND ;
     ( lnPrimaryKeyFieldState = 1 OR lnPrimaryKeyFieldState = 2)
   llOKToSet = .T.
CASE ( tnFieldState = 3 OR tnFieldState = 4 ) AND ;
```

```
     ( lnPrimaryKeyFieldState = 3 OR lnPrimaryKeyFieldState = 4)
   llOKToSet = .T.
ENDCASE
IF llOKToSet
   *-------------------------------------------------------------
   *--- Set the field state of the primary key to the specified value
   *-------------------------------------------------------------
   =SETFLDSTATE(lcPrimaryKey, tnFieldState)
ENDIF
```

The sole objective of this code is to change the primary key field state to the specified value. Some error checking code was added for good measure.

Conclusion

Well, this was a rather interesting situation. Codebook had been out for almost two years before this little bug was made public and fixed by Oakleaf. I find this to be extremely interesting considering that thousands of records of hundreds of companies were experiencing this problem on every save and no one found it until now.

Oakleaf Enterprises offers their own version of Codebook. Check out their web site http://www.oakleafsd.com/. Some of the specifics of this offer follows:

> For only $349 ($499 for multi-developer site license) you will receive four (4) updates over the next year via the Internet. If you would like to receive the updates via US mail, just add $25 for shipping & handling.

Charles T. Blankenship is president of Software Assets of Virginia, Inc (SAVI), a computer consulting firm that specializes in developing mission critical Visual FoxPro based applications using Codebook technology.. Phone: (757) 853-4465, Internet: ctb@savvysolutions.com, Web Site: ***www.savvysolutions.com****, CompuServe 76132,2575*

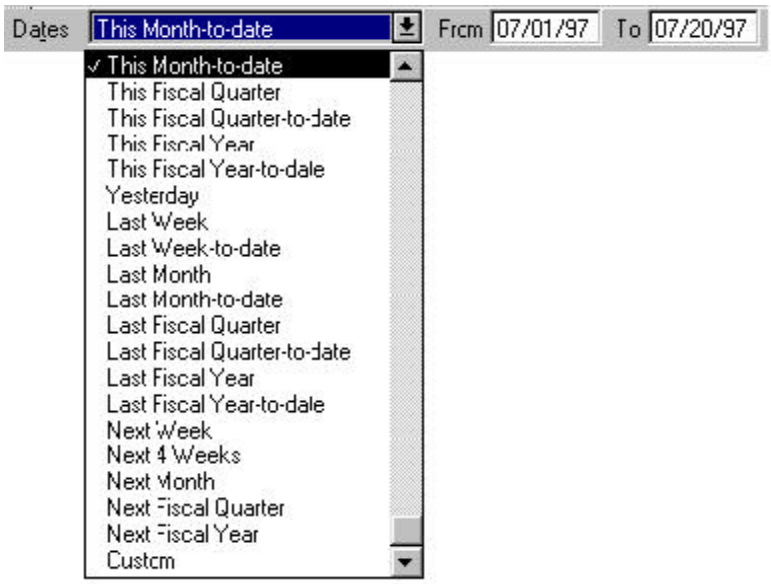
Quick Books Date Range Control

One of the best examples of interface design is Intuit's series of financial helpers like Quicken and Quickbooks. This article describes one very helpful control that allows a user to select a range of dates without forcing them to enter the actual From and To date range.

Introduction

Just about every application ever created requires the user to enter a range of dates, and just about every application makes this chore much more difficult than it needs to be. Intuit has developed a control that greatly simplifies this task. Now instead of entering a date range, the user is presented with a textual representation of the date range and the application provides the specified date values automatically. Providing this functionality to your users is very easy to do. The first section of this article describes how to use the SAVIContainedQBDateRangeControl that is provided with the SAVI Codebook Framework. The second half of the article explains how this class definition works so you can build it yourself if you are using some other variation of Codebook or plain Visual FoxPro.

Figure One: QuickBooks Date Range Control extracted from QuickBooks Version 5.0

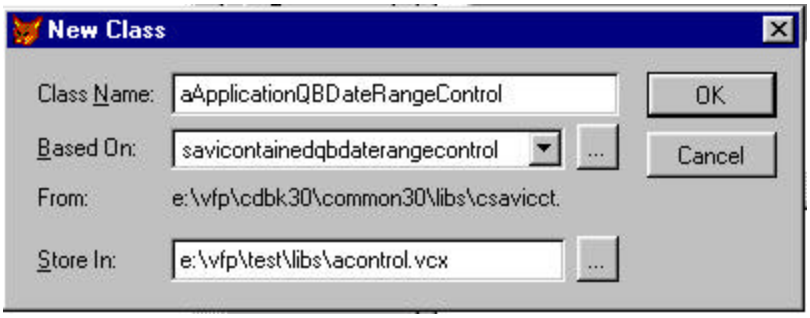


Using the SAVIContainedQBDateRangeControl in Development

1. Subclass the SAVIContainedQBDateRangeControl (which is located in \CDBK30\COMMON30\LIBS\SAVICCTL.VCX) into a new class named aApplicationQBDateRangeControl. Store this new class in a class library named aControl.VCX. Store the aControl.VCX in the LIBS subdirectory of the application. Name the new class aApplicationQBDateRangeControl. This new class, rather than the base class provided in the SAVICCtl.VCX, is now the superclass for all instances of this control in the application. To create this control, fill out the New Class dialog as illustrated:

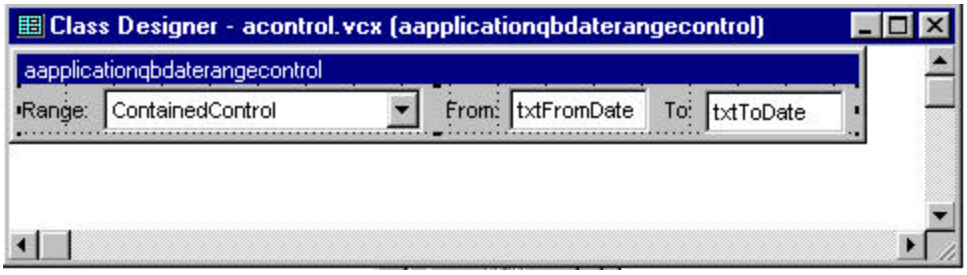
a. Bring up your application's project manager.
b. Press the New button to create a new class definition.
c. Fill out the New Class dialog as illustrated in Figure Two below.
d. Press the OK button.

Figure Two: New Class Dialog for subclassing SAVIContainedQBDateRangeControl



2. After you press the OK button on the New Class dialog, the Visual Class Designer appears as illustrated below. Bring up the property sheet for this control and locate the custom property named **`.cMMDDFiscalYearBegins`**, it is located on the Other tab of the property sheet. If your client's fiscal year begins on a day other than January 1, enter the month and day on which their fiscal year does begin. For example, if their fiscal year begins on March 1, enter "**`03/01/`**"for this property. The format of this information is critical. Notice that there is a forward slash between the MM and the DD as well as a forward slash following the DD component. Ensure your information follows these rules. The reason for subclassing this control is that not all of your clients will have the same fiscal year. Therefore, placing a client's fiscal year information in your *framework* superclass is not appropriate. Subclass this control and override the properties with the information needed to make the control work with your current client and store this class in *their* class libraries. In Codebook, a client's class libraries are located in the LIBS subdirectory of the application.

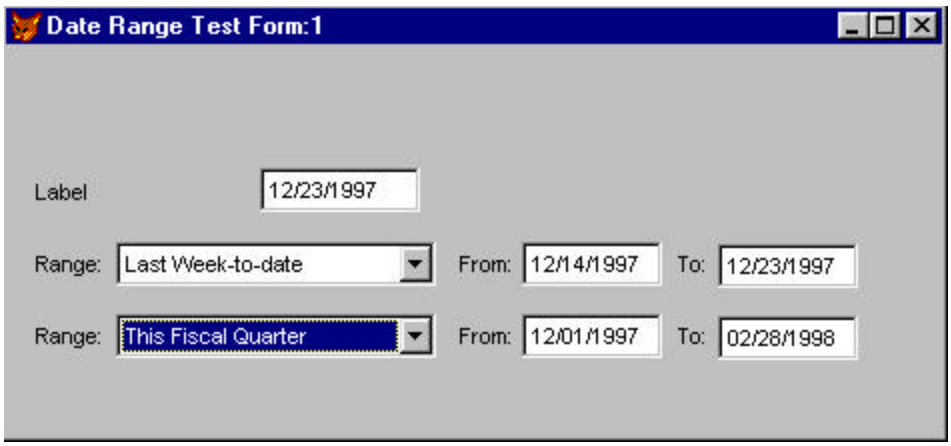
Figure Three: Class Designer for aApplicationQBDateRangeControl



3. Next, close this dialog and save your changes. The new class library, aControl.VCX, that contains your new date range control superclass should appear in your project manager.
4. Finally, drag and drop this control onto the business object that requires it and modify the **`Label.Caption`** property to indicate what type of date range this control is providing. For

example, if this control is used to accept dates for a Profit and Loss report, make the **`Label.Caption`** contain **`P & L Date Range`**. You also have an option of subclassing this control again and placing the caption information in the new subclass. Encapsulating control specific information in its own subclass allows you to use the ProfitLossDateRangeControl without having to make modifications directly on the business object where it is used. This is a matter of taste, however. Do what makes you the most comfortable.

Now, your users will be able to select date ranges based upon a descriptive name instead of having to remember that the last fiscal quarter began on … what date was that? The following illustration is a Date Range Test Form that contains two date range controls. Each illustrate a different textual date range description and their corresponding, automatically calculated, date ranges.



The final step when using this control in application development is extracting the actual dates that have been calculated based upon the user's request. The From and the To dates are stored in two properties that are members of the control container. The From date is stored in a property named **`.dFromDate`**, the To Date is stored in a property name **`.dToDate`**. Use these properties to extract the date ranges from the control so you can use them in your processing.

Creating the Control Yourself

The Components

This control is composed, primarily of a container object. The container contains a ComboBox, two TextBoxes, and three Label objects. The ComboBox displays the possible selections for date ranges and presents them in a textual format. The two text boxes, one of which contains a from date, the other contains a to date, display the calculated values based upon the user's date range selection.

The Functionality

The first step requires determining the list of possible choices the users can pick. The following is a list of options for this control. These options were extracted (read stolen) from the Quick Books control of the same type.

The next step is to determine how the control should calculate the date range when the user selects the desired value. To accomplish this, I decided to create a method for each selection. The available selections and their corresponding methods are listed below.

All	GetAll( )
Today	GetToday( )
This Week	GetThisWeek( )
This Week-to-date	GetThisWeekToDate( )
This Month	GetThisMonth( )
This Month-to-date	GetThisMonthToDate( )
This Fiscal Quarter	GetThisFiscalQuarter( )
This Fiscal Quarter-to-date	GetThisFiscalQuarterToDate( )
This Fiscal Year	GetThisFiscalYear( )
This Fiscal Year-to-date	GetThisFiscalYearToDate( )
Yesterday	GetYesterday( )
Last Week	GetLastWeek( )
Last Week-to-date	GetLastWeekToDate( )
Last Month	GetLastMonth( )
Last Month-to-date	GetLastMonthToDate( )
Last Fiscal Quarter	GetLastFiscalQuarter( )
Last Fiscal Quarter-to-date	GetLastFiscalQuarterToDate( )
Last Fiscal Year	GetLastFiscalYear( )
Last Fiscal Year-to-date	GetLastfiscalYearToDate( )
Next Week	GetNextWeek( )
Next 4 Weeks	GetNextFourWeeks( )
Next Month	GetNextMonth( )
Next Fiscal Quarter	GetNextFiscalQuarter( )
Next Fiscal Year	GetNextFiscalYear( )
Custom	GetCustom( )
January – YYYY	GetMonth("01")
February – YYYY	GetMonth("02")
March – YYYY	GetMonth("03")
April – YYYY	GetMonth("04")
May – YYYY	GetMonth("05")
June –YYYY	GetMonth("06")
July – YYYY	GetMonth("07")
August – YYYY	GetMonth("08")
September – YYYY	GetMonth("09")
October – YYYY	GetMonth("10")
November – YYYY	GetMonth("11")
December – YYYY	GetMonth("12")

This approach may seem excessive at first glance, however, following the principles of programming in the problem domain ensured that there were only about five or six functions that performed date calculations. The remaining methods simply called the needed functionality to

accomplish the desired task. In this case, programming in the problem domain allowed for a high degree of reuse.

The Initialization

The following is the code present in the Init() method of the container.

Listing One: SAVIContainedQBDateRangeConrol::Init() method

```
LOCAL llRetVal

llRetVal = SAVIContainedComboBox::Init()
llRetVal = llRetVal AND THIS.SetReferenceYear()
llRetVal = llRetVal AND THIS.PopulateDateRangeComboBox()
llRetVal = llRetVal AND THIS.SetReferenceDate()
llRetVal = llRetVal AND THIS.GetFiscalYearBeginDate()

RETURN llRetVal
```

The first step in this process is to call superclass functionality. The next step is to determine the year in which this operation is taking place. The next step is to populate the contained combo box with the list of choices the user can make. Next, the control must retrieve the date against which all date calculations are made. Finally, the control must identify the date on which the fiscal year begins.

Sidebar One: One of Codebook's MANY Values

I must take a second here to illustrate a very important point. The contribution to the FoxPro community by Yair Alan Griver's Flash Creative Management cannot be underestimated. For $40.00, they made available a veritable treasure trove of "How To" information. Mac Rubel, when I asked him about Codebook several years ago, described it as "the deal of the century". He obviously is a master of the understatement. There are examples buried in this framework that boggles the mind of the new VFP developer. Several issues ago, Tamar Granor, in FoxPro Advisor, advised users to look at the sample applications provided with the VFP product and use those examples as a "How To" to accomplish development tasks. Codebook, with its rock bottom price, serves this purpose much better than the sample applications provided with VFP. Therefore, if you are interested in becoming a better developer, buy Codebook and study it. It is the highest quality, and most cheaply priced "How To" available on the market today.

Setting the Reference Year

Setting the reference year is as easy as using the YEAR() function, converting its return value to a character and trimming it up for use. The name of the method is SetReferenceYear(). It is a member of the SAVIContianedQBDateRangeControl, and its code looks like this:

Listing Two: SAVIContainedQBDateRangeControl::SetReferenceYear( )

```
THIS.cYear = ALLTRIM(STR(YEAR( DATE() )))
```

Notice that a member property, named **`cYear`**, is where the year information is stored.

Populating the Date Range ComboBox

The technique used for accomplishing this task was *stolen*, directly, from the cStateComboBox::Init() method located in the cCustCtl.VCX of the original Codebook. Two requirements had to be met when populating the combo box. The first involved storing the date range text that must be displayed to the user. The second involved storing the name of the method to be executed when the user makes their selection. The following code ensures the required information is stored in the combo box. Place this code in a method named **`PopulateDateRangeComboBox().`** Ensure that the aforementioned method is a member of the container that holds the ComboBox.

Listing Two: SAVIContainedQBDateRangeConrol::PopulateDateRangeComboBox() method

```
LOCAL lcYear
lcYear = THIS.cYear

THIS.ContainedControl.AddListItem("All"                                    , 1, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetAll()"                   , 1, 2)

THIS.ContainedControl.AddListItem("Today"                                  , 2, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetToday()"                 , 2, 2)

THIS.ContainedControl.AddListItem("This Week"                              , 3, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisWeek()"              , 3, 2)

THIS.ContainedControl.AddListItem("This Week-to-date"                      , 4, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisWeekToDate()"        , 4, 2)

THIS.ContainedControl.AddListItem("This Month"                             , 5, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisMonth()"             , 5, 2)

THIS.ContainedControl.AddListItem("This Month-to-date"                     , 6, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisMonthToDate()"       , 6, 2)

THIS.ContainedControl.AddListItem("This Fiscal Quarter"                    , 7, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalQuarter()"     , 7, 2)

THIS.ContainedControl.AddListItem("This Fiscal Quarter-to-date"            , 8, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalQuarterToDate()" , 8, 2)

THIS.ContainedControl.AddListItem("This Fiscal Year"                       , 9, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalYear()"        , 9, 2)

THIS.ContainedControl.AddListItem("This Fiscal Year-to-date"               , 10, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalYearToDate()"  , 10, 2)

THIS.ContainedControl.AddListItem("Yesterday"                              , 11, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetYesterday()"             , 11, 2)
```

```
THIS.ContainedControl.AddListItem("Last Week"                                , 12, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastWeek()"                , 12, 2)

THIS.ContainedControl.AddListItem("Last Week-to-date"                        , 13, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastWeekToDate()"          , 13, 2)

THIS.ContainedControl.AddListItem( "Last Month"                              , 14, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastMonth()"              , 14, 2)

THIS.ContainedControl.AddListItem( "Last Month-to-date"                      , 15, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastMonthToDate()"        , 15, 2)

THIS.ContainedControl.AddListItem( "Last Fiscal Quarter"                     , 16, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastFiscalQuarter()"      , 16, 2)

THIS.ContainedControl.AddListItem( "Last Fiscal Quarter-to-date"             , 17, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastFiscalQuarterToDate()", 17, 2)

THIS.ContainedControl.AddListItem("Last Fiscal Year"                         , 18, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastFiscalYear()"          , 18, 2)

THIS.ContainedControl.AddListItem("Last Fiscal Year-to-date"                 , 19, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastFiscalYearToDate()"    , 19, 2)

THIS.ContainedControl.AddListItem("Next Week"                                , 20, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextWeek()"                , 20, 2)

THIS.ContainedControl.AddListItem("Next Four Weeks"                          , 21, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFourWeeks()"           , 21, 2)

THIS.ContainedControl.AddListItem("Next Month"                               , 22, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextMonth()"               , 22, 2)

THIS.ContainedControl.AddListItem("Next Fiscal Quarter"                      , 23, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFiscalQuarter()"       , 23, 2)

THIS.ContainedControl.AddListItem("Next Fiscal Year"                         , 24, 1)
THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFiscalYear()"          , 24, 2)

THIS.ContainedControl.AddListItem( "Custom "                                 , 25, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetCustom()"                 , 25, 2)

THIS.ContainedControl.AddListItem( "January - " + lcYear        , 26, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('01')" , 26, 2)

THIS.ContainedControl.AddListItem( "February - " + lcYear       , 27, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('02')" , 27, 2)

THIS.ContainedControl.AddListItem( "March - " + lcYear          , 28, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('03')" , 28, 2)
```

```
THIS.ContainedControl.AddListItem( "April - " + lcYear          , 29, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('04')" , 29, 2)

THIS.ContainedControl.AddListItem( "May - " + lcYear            , 30, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('05')" , 30, 2)

THIS.ContainedControl.AddListItem( "June - " + lcYear           , 31, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('06')" , 31, 2)

THIS.ContainedControl.AddListItem( "July - " + lcYear           , 32, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('07')" , 32, 2)

THIS.ContainedControl.AddListItem( "August - " + lcYear         , 33, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('08')" , 33, 2)

THIS.ContainedControl.AddListItem( "September - " + lcYear      , 34, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('09')" , 34, 2)

THIS.ContainedControl.AddListItem( "October - " + lcYear        , 35, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('10')" , 35, 2)

THIS.ContainedControl.AddListItem( "November - " + lcYear       , 36, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('11')" , 36, 2)

THIS.ContainedControl.AddListItem( "December - " + lcYear       , 37, 1)
THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('12')" , 37, 2)
```

Setting the Reference Date

The next chore is to capture the date against which all date calculations are to be made. This is accomplished by a method named **`SetReferenceDate( )`** which is also a member of the control container. A member property named **`dReferenceDate`** stores the date value. The code for this method looks like this:

Listing Three: SAVIContainedQBDateRangeControl::SetReferenceDate() method

```
THIS.dReferenceDate = DATE()
```

Notice that this is a one-line method. It is important to remember that methods are not required to have 10, 20 or 30 lines of code. Every method you create should accomplish one discrete task. If it takes one line of code to accomplish that task, then so be it. If your method takes 30 lines of code, start examining the method code for areas where it can be broken down and simplified. The important concept to remember here is that the task that is accomplished may be required several different times. IMPORTANT: The only way to achieve reuse is to ensure that the methods you create *can be* easily reused. The only way to accomplish *that* is to make sure your methods do one and *only* one thing. This method does one and only one thing.

Getting the Fiscal Year Begin Date

This is the only date information that must be provided by the developer before the control can become fully operational. All Fiscal Year calculations must be based on the day of the year on which your client's fiscal year begins. The code for this method looks like this:

Listing Four: SAVIContainedQBDateRangeControl::GetFiscalYearBeginDate()

```
LOCAL llRetVal
llRetVal = .T.

IF .NOT. EMPTY( THIS.cMMDDFiscalYearBegins )
   ldFiscalYearBeginDate = CTOD( THIS.cMMDDFiscalYearBegins + THIS.cYear )
ELSE
   ldFiscalYearBeginDate = CTOD( "01/01/" + THIS.cYear )
ENDIF

IF EMPTY( THIS.dFiscalYearBeginDate )
   THIS.dFiscalYearBeginDate = ldFiscalYearBeginDate
ENDIF

IF TYPE( "THIS.dFiscalYearBeginDate" ) == "D" AND ;
  .NOT. EMPTY( THIS.dFiscalYearBeginDate )
   THIS.CalculateFiscalQuarterBeginDates()
ELSE
   llRetVal = .F.
ENDIF

RETURN llRetVal
```

This code begins by combining the month and the day the fiscal year begins with the current year and converting that information to a date type value. If the developer provided the month and day their client's fiscal year begins, that information is used. If not, it is assumed that January 1, is the fiscal year begin date. Once the date value has been calculated, the member property, **`dFiscalYearBeginDate,`** is tested to make sure it is empty. If it is, the newly calculated fiscal year begin date is stored there. If a valid date has been provided, the begin dates of each of the remaining fiscal quarters are then calculated.

Calculating Fiscal Quarter Begin Dates

Once this fiscal year begin date is known, it is a relatively trivial matter to calculate the begin dates of the remaining three fiscal quarters. Simply take the fiscal year begin date, and, using the GOMONTH() function, determine what the date is three, six and nine months in the future. Store the resulting dates in the specified member properties.

Listing Five: SAVIContainedQBDateRangeControl::CalculateFiscalQuarterBeginDates( )

```
THIS.dSecondFiscalQuarterBeginDate = GOMONTH( THIS.dFiscalYearBeginDate, 3 )
THIS.dThirdFiscalQuarterBeginDate  = GOMONTH( THIS.dFiscalYearBeginDate, 6 )
THIS.dFourthFiscalQuarterBeginDate = GOMONTH( THIS.dFiscalYearBeginDate, 9 )
```

The Interactive Element

Once the ComboBox is populated with options, the user can make their selection. Since the InteractiveChange event is triggered each time the user selects an item from the ComboBox, this event is the logical location to place the required code, illustrated below:

Listing Six: ComboBox::InteractiveChange() method

```
LOCAL lcCommand , ;
      lnListItemIndex

*-------------------------------------------------------------
*-- Ensure the date information is updated.  Someone may have
*-- been working past midnight.  Some poor accountant may
*-- be working past midnight on December 31!
*-------------------------------------------------------------
THIS.PARENT.SetReferenceYear()
THIS.PARENT.SetReferenceDate()

lnListItemIndex = THIS.ListIndex
lcCommand       = THIS.ListItem( lnListItemIndex, 2 )
&lcCommand

THIS.PARENT.PopulateFromToDateControls()
```

When the user selects the desired date range from the ComboBox, the index of the selected date range is determined as follows:

```
lnListItemIndex = THIS.ListIndex
```

This value is then used to extract the name of the method that should be executed. The name of the method is located in the second column of the ComboBox control and is determined as follows:

```
lcCommand = THIS.ListItem( lnListItemIndex, 2 )
```

Next, the specified method is executed using macro substitution.

```
&lcCommand
```

Finally, the values derived by the specified method must be transferred to the contained From and To date controls. This is the responsibility of the **`PopulateFromToDateControls()`** method.

Displaying the Calculated Dates

The functionality of the **`PopulateFromToDateControls()`** method is very simple. The methods that calculate the date ranges place the results of their calculations into the following member properties, **`.dFromDate`** and **`.dToDate`**. In order for the user to view the dates, they must be transferred to the contained controls that display them to the user. This is accomplished as follows:

Listing Seven: SAVIContianedQBDateControl::PopulateFromToDateControls()

```
THIS.txtFromDate.Value = THIS.dFromDate
THIS.txtToDate.Value   = THIS.dToDate
```

The Core Date Calculation Processes

There are four core date calculations that perform most of the work of the control. All of them accept a date parameter upon which the calculation is based. All of the methods that deal with week calculations assume that the developer has specified what the first day of the week is by properly populating the **`.nWeekBeginDefault`** property. See FoxPro help under the DOW( ) function for the valid values you can place in this property. This allows you to specify which day of the week begins *your* week.

Getting the First Day of the Week

The first step in this process is to take the date, passed in by parameter, and determine the day of the week represented by that date. For example, if you passed in the date December 24, 1997, and Sunday was the beginning of your week, VFP would return the number 4. Sunday is first, Monday is second, Tuesday is third and Wednesday is the fourth. The object of this game is to determine the date that represents Sunday, which was three days ago. Subtracting one from the value returned from DOW and subtracting this result from the date passed in by parameter will always yield the date that represents the beginning of your week.

Listing Eight: SAVIContainedQBDateRangeControl::GetFirstDayOfWeek(<dDate>)

```
LPARAMETERS tdDate

LOCAL lnDayOfWeek

lnDayOfWeek    = DOW(tdDate, THIS.nWeekBeginDefault)
THIS.dFromDate = tdDate - (lnDayOfWeek - 1)
```

Getting the Last Day of the Week

This process requires determining the date that represents the last day of your week. To accomplish this the date passed in by parameter is passed as a parameter to the DOW() function. This function returns a numeric representation of this date. Since all weeks consist of seven days, the last day of the week can be derived by subtracting the current date's DOW() value from seven and adding that number to the current day.

For example, if you passed December 24, 1997 to this control, DOW() returns the number four. Subtracting four from seven yields three. Adding three days to December 24, 1997 gives Saturday, the last day of the week.

Listing Nine: SAVIContainedQBDateRangeControl::GetLastDayOfWeek(<dDate>)

```
LPARAMETERS tdDate

LOCAL lnDayOfWeek
```

```
lnDayOfWeek    = DOW(tdDate, THIS.nWeekBeginDefault)
THIS.dToDate = tdDate + ( 7 - lnDayOfWeek )
```

Getting the First Day of the Month

This is the simplest calculation of all. The only thing that must be calculated is the day number of the current date (passed in by parameter). Subtracting one from this value yields the total number of days to the first day of the month. Subtracting that value from the current date yields the first day of the month.

Listing Ten: SAVIContainedQBDateRangeControl::GetFirstDayOfMonth(<dDate>)

```
LPARAMETERS tdDate

THIS.dFromDate = tdDate - ( DAY( tdDate) - 1 )
```

Getting the Last Day of the Month

To calculate the last day of *any* month, the only thing you have to do is use the same calculation used above to find the first day of the month, use the GOMONTH() function to find the first day of the NEXT month, and finally, subtract one from that to determine the last day of THIS month.

Listing Eleven: SAVIContainedQBDateRangeControl::GetLastDayOfMonth(<dDate>)

```
LPARAMETER tdDate

LOCAL ldFirstDayOfMonth
ldFirstDayOfMonth = tdDate - ( DAY( tdDate) - 1 )

THIS.dToDate = GOMONTH( ldFirstDayOfMonth, 1 ) - 1
```

The remainder of the date calculations are included in the class definition listed below.

SAVIContainedQBDateRangeControl Class Listing

```
**********************************************
*-- Class:        savicontainedqbdaterangecontrol
(e:\vfp\cdbk30\common30\libs\csavicct.vcx)
*-- ParentClass:  savicontainedcombobox (e:\vfp\cdbk30\common30\libs\csavicct.vcx)
*-- BaseClass:    container
*-- Quick Books Date Range Text Box
*
DEFINE CLASS savicontainedqbdaterangecontrol AS savicontainedcombobox



	PROTECTED cyear
	Width = 421
	Height                       = 26
	cyear                        = ""
	dfromdate                    = {  /  /  }
	dtodate                      = {  /  /  }
	nweekbegindefault            = 1
	dreferencedate               = {  /  /  }
	dfiscalyearbegindate         = {  /  /  }
	cmmddfiscalyearbegins        = "01/01/"
	dsecondfiscalquarterbegindate = {  /  /  }
	dthirdfiscalquarterbegindate  = {  /  /  }
	dfourthfiscalquarterbegindate = {  /  /  }
	Name                         = "savicontainedqbdaterangecontrol"
	Label.Caption                = "Range:"
	Label.Name                   = "Label"
	ContainedControl.ColumnCount  = 2
	ContainedControl.ColumnWidths = "140,0"
	ContainedControl.Height       = 22
	ContainedControl.ColumnLines  = .F.
	ContainedControl.Left         = 43
	ContainedControl.Top          = 2
	ContainedControl.Width        = 160
	ContainedControl.Name         = "ContainedControl"


	ADD OBJECT txtfromdate AS savidatetextbox WITH ;
		Alignment = 3, ;
		Value     = {  /  /  }, ;
		Height    = 22, ;
		Left      = 246, ;
		Top       = 2, ;
		Width     = 70, ;
		lbound    = .F., ;
		Name      = "txtFromDate"


	ADD OBJECT txttodate AS savidatetextbox WITH ;
		Alignment = 3, ;
```

```
        Value     = {  /  /  }, ;
        Height    = 22, ;
        Left      = 344, ;
        Top       = 3, ;
        Width     = 70, ;
        lbound    = .F., ;
        Name      = "txtToDate"


    ADD OBJECT lblfrom AS savilabel WITH ;
        Caption = "From:", ;
        Left    = 215, ;
        Top     = 6, ;
        Name    = "lblFrom"


    ADD OBJECT lblto AS savilabel WITH ;
        Caption = "To:", ;
        Left    = 324, ;
        Top     = 6, ;
        Name    = "lblTo"


    *-- PROTECTED - populates the date range ComboBox with the specified date ranges.
    PROTECTED PROCEDURE populatedaterangecombobox
        LOCAL lcYear
        lcYear = THIS.cYear

        THIS.ContainedControl.AddListItem("All"                                    ,
1, 1)
        THIS.ContainedControl.AddListItem("THIS.PARENT.GetAll()"                   ,
1, 2)

        THIS.ContainedControl.AddListItem("Today"                                  ,
2, 1)
        THIS.ContainedControl.AddListItem("THIS.PARENT.GetToday()"                 ,
2, 2)

        THIS.ContainedControl.AddListItem("This Week"                              ,
3, 1)
        THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisWeek()"              ,
3, 2)

        THIS.ContainedControl.AddListItem("This Week-to-date"                      ,
4, 1)
        THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisWeekToDate()"        ,
4, 2)

        THIS.ContainedControl.AddListItem("This Month"                             ,
5, 1)
```

```
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisMonth()"              ,
5, 2)

            THIS.ContainedControl.AddListItem("This Month-to-date"                      ,
6, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisMonthToDate()"        ,
6, 2)

            THIS.ContainedControl.AddListItem("This Fiscal Quarter"                     ,
7, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalQuarter()"      ,
7, 2)

            THIS.ContainedControl.AddListItem("This Fiscal Quarter-to-date"             ,
8, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalQuarterToDate()",
8, 2)

            THIS.ContainedControl.AddListItem("This Fiscal Year"                        ,
9, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalYear()"         ,
9, 2)

            THIS.ContainedControl.AddListItem("This Fiscal Year-to-date"                ,
10, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetThisFiscalYearToDate()"   ,
10, 2)

            THIS.ContainedControl.AddListItem("Yesterday"                               ,
11, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetYesterday()"              ,
11, 2)

            THIS.ContainedControl.AddListItem("Last Week"                               ,
12, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastWeek()"               ,
12, 2)

            THIS.ContainedControl.AddListItem("Last Week-to-date"                       ,
13, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastWeekToDate()"         ,
13, 2)

            THIS.ContainedControl.AddListItem( "Last Month"                             ,
14, 1)
            THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastMonth()"             ,
14, 2)

            THIS.ContainedControl.AddListItem( "Last Month-to-date"                     ,
15, 1)
            THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastMonthToDate()"       ,
15, 2)
```

```
            THIS.ContainedControl.AddListItem( "Last Fiscal Quarter"                    ,
16, 1)
            THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastFiscalQuarter()"     ,
16, 2)

            THIS.ContainedControl.AddListItem( "Last Fiscal Quarter-to-date"            ,
17, 1)
            THIS.ContainedControl.AddListItem( "THIS.PARENT.GetLastFiscalQuarterToDate()",
17, 2)

            THIS.ContainedControl.AddListItem("Last Fiscal Year"                        ,
18, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastFiscalYear()"         ,
18, 2)

            THIS.ContainedControl.AddListItem("Last Fiscal Year-to-date"                ,
19, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetLastFiscalYearToDate()"   ,
19, 2)

            THIS.ContainedControl.AddListItem("Next Week"                               ,
20, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextWeek()"               ,
20, 2)

            THIS.ContainedControl.AddListItem("Next Four Weeks"                         ,
21, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFourWeeks()"          ,
21, 2)

            THIS.ContainedControl.AddListItem("Next Month"                              ,
22, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextMonth()"              ,
22, 2)

            THIS.ContainedControl.AddListItem("Next Fiscal Quarter"                     ,
23, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFiscalQuarter()"      ,
23, 2)

            THIS.ContainedControl.AddListItem("Next Fiscal Year"                        ,
24, 1)
            THIS.ContainedControl.AddListItem("THIS.PARENT.GetNextFiscalYear()"         ,
24, 2)

            THIS.ContainedControl.AddListItem( "Custom "                                ,
25, 1)
            THIS.ContainedControl.AddListItem( "THIS.PARENT.GetCustom()"                ,
25, 2)

            THIS.ContainedControl.AddListItem( "January - " + lcYear        , 26, 1)
```

```
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('01')" , 26, 2)

        THIS.ContainedControl.AddListItem( "February - " + lcYear       , 27, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('02')" , 27, 2)

        THIS.ContainedControl.AddListItem( "March - " + lcYear          , 28, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('03')" , 28, 2)

        THIS.ContainedControl.AddListItem( "April - " + lcYear          , 29, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('04')" , 29, 2)

        THIS.ContainedControl.AddListItem( "May - " + lcYear            , 30, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('05')" , 30, 2)

        THIS.ContainedControl.AddListItem( "June - " + lcYear           , 31, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('06')" , 31, 2)

        THIS.ContainedControl.AddListItem( "July - " + lcYear           , 32, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('07')" , 32, 2)

        THIS.ContainedControl.AddListItem( "August - " + lcYear         , 33, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('08')" , 33, 2)

        THIS.ContainedControl.AddListItem( "September - " + lcYear      , 34, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('09')" , 34, 2)

        THIS.ContainedControl.AddListItem( "October - " + lcYear        , 35, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('10')" , 35, 2)

        THIS.ContainedControl.AddListItem( "November - " + lcYear       , 36, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('11')" , 36, 2)

        THIS.ContainedControl.AddListItem( "December - " + lcYear       , 37, 1)
        THIS.ContainedControl.AddListItem( "THIS.PARENT.GetMonth('12')" , 37, 2)
ENDPROC


PROCEDURE gettoday
        THIS.dFromDate = THIS.dReferenceDate
        THIS.dToDate   = THIS.dReferenceDate
ENDPROC


PROCEDURE getthisweek
        THIS.GetFirstDayOfWeek( THIS.dReferenceDate )
        THIS.GetLastDayOfWeek( THIS.dReferenceDate )
ENDPROC


PROCEDURE getthisweektodate
        THIS.GetFirstDayOfWeek( THIS.dReferenceDate )
        THIS.dToDate = THIS.dReferenceDate
```

```
ENDPROC


PROCEDURE getthismonth
	lcMonth = ALLTRIM(STR(MONTH( THIS.dReferenceDate )))
	THIS.GetMonth( lcMonth )
ENDPROC


PROCEDURE getthismonthtodate
	THIS.GetFirstDayOfMonth( THIS.dReferenceDate )
	THIS.dToDate = THIS.dReferenceDate
ENDPROC


PROCEDURE getthisfiscalquartertodate
	THIS.GetThisFiscalQuarter( .T. )
	THIS.dToDate = THIS.dReferenceDate
ENDPROC


PROCEDURE getthisfiscalyear
	THIS.dFromDate = THIS.dFiscalYearBeginDate
	THIS.dToDate = GOMONTH( THIS.dFiscalYearBeginDate, 12 ) - 1
ENDPROC


PROCEDURE getthisfiscalyeartodate
	THIS.dFromDate = THIS.dFiscalYearBeginDate
	THIS.dToDate   = THIS.dReferenceDate
ENDPROC


PROCEDURE getyesterday
	THIS.dFromDate = THIS.dReferenceDate - 1
	THIS.dToDate   = THIS.dReferenceDate - 1
ENDPROC




PROCEDURE getlastweek
	THIS.GetFirstDayOfWeek( THIS.dReferenceDate - 7 )
	THIS.GetLastDayOfWeek( THIS.dReferenceDate - 7 )
ENDPROC


PROCEDURE getlastweektodate
	THIS.GetFirstDayOfWeek( THIS.dReferenceDate - 7 )
	THIS.dToDate = THIS.dReferenceDate
ENDPROC
```

```
PROCEDURE getlastmonth
       LPARAMETERS tlBeginDateOnly
       LOCAL ldLastDayOfLastMonth

       ldLastDayOfLastMonth = THIS.dReferenceDate - DAY( THIS.dReferenceDate )
       THIS.GetFirstDayOfMonth( ldLastDayOfLastMonth )
       IF .NOT. tlBeginDateOnly
          THIS.GetLastDayOfMonth( ldLastDayOfLastMonth )
       ENDIF
ENDPROC


PROCEDURE getlastmonthtodate
       THIS.GetLastMonth( .T. )
       THIS.dToDate = THIS.dReferenceDate
ENDPROC


PROCEDURE getlastfiscalquarter
       LPARAMETERS tlBeginDateOnly

       LOCAL ldFromDate, ;
             ldToDate

       DO CASE
       CASE BETWEEN( THIS.dReferenceDate, ;
                     THIS.dFiscalYearBeginDate, ;
                     THIS.dSecondFiscalQuarterBeginDate - 1 )

          ldFromDate = GOMONTH( THIS.dFiscalYearBeginDate, -3 )
          ldToDate   = THIS.dFiscalYearBeginDate - 1

       CASE BETWEEN( THIS.dReferenceDate, ;
                     THIS.dSecondFiscalQuarterBeginDate, ;
                     THIS.dThirdFiscalQuarterBeginDate - 1)

          ldFromDate = THIS.dFiscalYearBeginDate
          ldToDate   = THIS.dSecondFiscalQuarterBeginDate - 1

       CASE BETWEEN( THIS.dReferenceDate, ;
                     THIS.dThirdFiscalQuarterBeginDate, ;
                     THIS.dFourthFiscalQuarterBeginDate - 1)

          ldFromDate = THIS.dSecondFiscalQuarterBeginDate
          ldToDate   = THIS.dThirdFiscalQuarterBeginDate - 1

       CASE BETWEEN( THIS.dReferenceDate, ;
                     THIS.dFourthfiscalQuarterBeginDate, ;
                     GOMONTH( THIS.dFiscalYearBeginDate, 12) - 1 )
```

```
            ldFromDate = THIS.dThirdFiscalQuarterBeginDate
            ldToDate   = THIS.dFourthFiscalQuarterBeginDate - 1

        ENDCASE

        THIS.dFromDate = ldFromDate

        IF .NOT. tlBeginDateOnly
           THIS.dToDate = ldToDate
        ENDIF


ENDPROC


PROCEDURE getlastfiscalquartertodate
        THIS.GetLastFiscalQuarter( .T. )
        THIS.dToDate = THIS.dReferenceDate
ENDPROC


PROCEDURE getlastfiscalyear
        LPARAMETERS tlBeginDateOnly

        lcLastYear = ALLTRIM(STR( VAL( THIS.cYear ) - 1 ))
        THIS.dFromDate = CTOD( THIS.cMMDDFiscalYearBegins + lcLastYear )

        IF .NOT. tlBeginDateOnly
           THIS.dToDate = GOMONTH( THIS.dFromDate, 12 ) - 1
        ENDIF
ENDPROC


PROCEDURE getlastfiscalyeartodate
        THIS.GetLastFiscalYear( .T. )
        THIS.dToDate = THIS.dReferenceDate
ENDPROC


PROCEDURE getnextweek
        THIS.GetFirstDayOfWeek( THIS.dReferenceDate + 7 )
        THIS.GetLastDayOfWeek( THIS.dReferenceDate + 7 )
ENDPROC


PROCEDURE getnextfourweeks
        THIS.GetFirstDayOfWeek( THIS.dReferenceDate + 7 )
        THIS.GetLastDayOfWeek( THIS.dReferenceDate + (7*4) )
ENDPROC


PROCEDURE getnextmonth
```

```
        LOCAL ldFirstDayOfThisMonth , ;
              ldFirstDayOfNextMonth

        ldFirstDayOfThisMonth = THIS.dReferenceDate - ;
                              ( DAY( THIS.dReferenceDate ) - 1 )
        ldFirstDayOfNextMonth = GOMONTH( ldFirstDayOfThisMonth, 1 )

        THIS.GetFirstDayOfMonth( ldFirstDayOfNextMonth )
        THIS.GetLastDayOfMonth( ldFirstDayOfNextMonth )
ENDPROC


PROCEDURE getnextfiscalquarter
        LPARAMETERS tlBeginDateOnly

        LOCAL ldFromDate, ;
              ldToDate

        DO CASE
        CASE BETWEEN( THIS.dReferenceDate, ;
                      THIS.dFiscalYearBeginDate, ;
                      THIS.dSecondFiscalQuarterBeginDate - 1 )

           ldFromDate = THIS.dSecondFiscalQuarterBeginDate
           ldToDate   = THIS.dThirdFiscalQuarterBeginDate - 1

        CASE BETWEEN( THIS.dReferenceDate, ;
                      THIS.dSecondFiscalQuarterBeginDate, ;
                      THIS.dThirdFiscalQuarterBeginDate - 1)

           ldFromDate = THIS.dThirdFiscalQuarterBeginDate
           ldToDate   = THIS.dFourthFiscalQuarterBeginDate - 1

        CASE BETWEEN( THIS.dReferenceDate, ;
                      THIS.dThirdFiscalQuarterBeginDate, ;
                      THIS.dFourthFiscalQuarterBeginDate - 1)

           ldFromDate = THIS.dFourthfiscalQuarterBeginDate
           ldToDate   = GOMONTH( THIS.dFiscalYearBeginDate, 12) - 1

        CASE BETWEEN( THIS.dReferenceDate, ;
                      THIS.dFourthfiscalQuarterBeginDate, ;
                      GOMONTH( THIS.dFiscalYearBeginDate, 12) - 1 )

           ldFromDate = CTOD(THIS.cMMDDFiscalYearBegins + ;
                       ALLTRIM(STR(VAL(THIS.cYear) + 1)))
           ldtoDate   = GOMONTH( ldFromDate, 3 ) - 1

        ENDCASE

        THIS.dFromDate = ldFromDate
        THIS.dToDate = ldToDate
```

```
ENDPROC


PROCEDURE getnextfiscalyear
	LOCAL lcNextYear
	lcNextYear = ALLTRIM(STR( VAL( THIS.cYear ) + 1 ))
	THIS.dFromDate = CTOD( THIS.cMMDDFiscalYearBegins + lcNextYear )
	THIS.dToDate = GOMONTH( THIS.dFromDate, 12 ) - 1
ENDPROC


PROCEDURE getmonth
	LPARAMETERS tcMonth

	LOCAL ldMonth
	ldMonth = CTOD( tcMonth + "/01/" + THIS.cYear )
	THIS.GetFirstDayOfMonth( ldMonth )
	THIS.GetLastDayOfMonth( ldMonth )
ENDPROC



PROCEDURE getthisfiscalquarter
	LPARAMETERS tlBeginDateOnly

	LOCAL ldFromDate, ;
	      ldToDate

	DO CASE
	CASE BETWEEN( THIS.dReferenceDate, ;
	              THIS.dFiscalYearBeginDate, ;
	              THIS.dSecondFiscalQuarterBeginDate - 1 )

	   ldFromDate = THIS.dFiscalYearBeginDate
	   ldToDate   = THIS.dSecondFiscalQuarterBeginDate - 1

	CASE BETWEEN( THIS.dReferenceDate, ;
	              THIS.dSecondFiscalQuarterBeginDate, ;
	              THIS.dThirdFiscalQuarterBeginDate - 1)

	   ldFromDate = THIS.dSecondFiscalQuarterBeginDate
	   ldToDate   = THIS.dThirdFiscalQuarterBeginDate - 1

	CASE BETWEEN( THIS.dReferenceDate, ;
	              THIS.dThirdFiscalQuarterBeginDate, ;
	              THIS.dFourthFiscalQuarterBeginDate - 1)

	   ldFromDate = THIS.dThirdFiscalQuarterBeginDate
	   ldToDate   = THIS.dFourthFiscalQuarterBeginDate - 1

	CASE BETWEEN( THIS.dReferenceDate, ;
	              THIS.dFourthfiscalQuarterBeginDate, ;
```

```
                    GOMONTH( THIS.dFiscalYearBeginDate, 12) - 1 )

            ldFromDate = THIS.dFourthfiscalQuarterBeginDate
            ldToDate   = GOMONTH( THIS.dFiscalYearBeginDate, 12) - 1

        ENDCASE

        THIS.dFromDate = ldFromDate

        IF .NOT. tlBeginDateOnly
           THIS.dToDate = ldToDate
        ENDIF


ENDPROC

PROTECTED PROCEDURE getlastdayofmonth
        LPARAMETER tdDate

        LOCAL ldFirstDayOfMonth
        ldFirstDayOfMonth = tdDate - ( DAY( tdDate) - 1 )

        THIS.dToDate = GOMONTH( ldFirstDayOfMonth, 1 ) - 1
ENDPROC


PROCEDURE populatefromtodatecontrols
        THIS.txtFromDate.Value = THIS.dFromDate
        THIS.txtToDate.Value = THIS.dToDate
ENDPROC



PROCEDURE setreferencedate
        *-----------------------------------------------------
        *-- Setting the date that all of the methods use as a
        *-- reference date in this method allows for extensive
        *-- testing of this control.  If DATE() were used in
        *-- all of the methods, testing would not be easily
        *-- accomplished.
        *-----------------------------------------------------
        THIS.dReferenceDate = DATE()
ENDPROC


PROCEDURE getfirstdayofweek
        LPARAMETERS tdDate

        LOCAL lnDayOfWeek

        lnDayOfWeek    = DOW(tdDate, THIS.nWeekBeginDefault)
        THIS.dFromDate = tdDate - (lnDayOfWeek - 1)
```

```
ENDPROC


PROCEDURE getlastdayofweek
       LPARAMETERS tdDate

       LOCAL lnDayOfWeek

       lnDayOfWeek    = DOW(tdDate, THIS.nWeekBeginDefault)
       THIS.dToDate = tdDate + ( 7 - lnDayOfWeek )
ENDPROC


PROCEDURE getfirstdayofmonth
       LPARAMETERS tdDate

       THIS.dFromDate = tdDate - ( DAY( tdDate) - 1 )
ENDPROC


PROCEDURE getfiscalyearbegindate
       *--------------------------------------------------------------------
       *-- Default the fiscal year begin date to the first day of the year
       *-- but do not override the programmer's default, if provided.
       *--------------------------------------------------------------------
       LOCAL llRetVal
       llRetVal = .T.

       IF .NOT. EMPTY( THIS.cMMDDFiscalYearBegins )
          ldFiscalYearBeginDate = CTOD( THIS.cMMDDFiscalYearBegins + THIS.cYear )
       ELSE
          ldFiscalYearBeginDate = CTOD( "01/01/" + THIS.cYear )
       ENDIF

       IF EMPTY( THIS.dFiscalYearBeginDate )
          THIS.dFiscalYearBeginDate = ldFiscalYearBeginDate
       ENDIF

       IF TYPE( "THIS.dFiscalYearBeginDate" ) == "D" AND ;
  .NOT. EMPTY( THIS.dFiscalYearBeginDate )
          THIS.CalculateFiscalQuarterBeginDates()
       ELSE
          llRetVal = .F.
       ENDIF

       RETURN llRetVal
ENDPROC


PROCEDURE setreferenceyear
       THIS.cYear = ALLTRIM(STR(YEAR( DATE() )))
ENDPROC
```

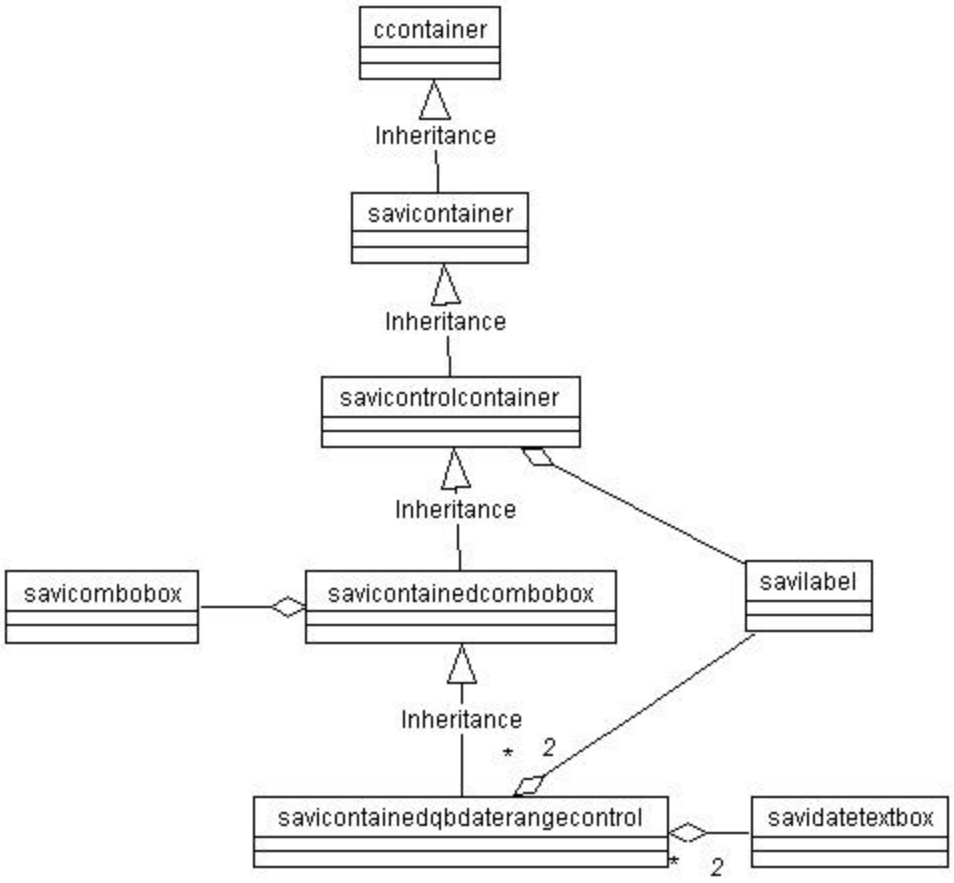
```
	PROCEDURE Init
		LOCAL llRetVal

		llRetVal = SAVIContainedComboBox::Init()
		llRetVal = llRetVal AND THIS.SetReferenceYear()
		llRetVal = llRetVal AND THIS.PopulateDateRangeComboBox()
		llRetVal = llRetVal AND THIS.SetReferenceDate()
		llRetVal = llRetVal AND THIS.GetFiscalYearBeginDate()

		RETURN llRetVal

	ENDPROC


	PROCEDURE calculatefiscalquarterbegindates
		THIS.dSecondFiscalQuarterBeginDate = GOMONTH( THIS.dFiscalYearBeginDate, 3
)
		THIS.dThirdFiscalQuarterBeginDate  = GOMONTH( THIS.dFiscalYearBeginDate, 6
)
		THIS.dFourthFiscalQuarterBeginDate = GOMONTH( THIS.dFiscalYearBeginDate, 9
)
	ENDPROC


	PROCEDURE ContainedControl.InteractiveChange
		LOCAL lcCommand , ;
			lnListItemIndex

		*-------------------------------------------------------------
		*-- Ensure the date information is updated.  Someone may have
		*-- been working past midnight.  Some poor accountant may
		*-- be working past midnight on December 31.
		*-------------------------------------------------------------
		THIS.PARENT.SetReferenceYear()
		THIS.PARENT.SetReferenceDate()

		lnListItemIndex = THIS.ListIndex
		lcCommand       = THIS.ListItem( lnListItemIndex, 2 )
		&lcCommand
		THIS.PARENT.PopulateFromToDateControls()
	ENDPROC

	PROCEDURE getall
	ENDPROC

	PROCEDURE getcustom
	ENDPROC


ENDDEFINE
```

```
*
*-- EndDefine: savicontainedqbdaterangecontrol
************************************************
```

The SAVI Contained Quick Books Date Range Control Class Diagram

The following is a class diagram for the SAVIContainedQBDateRangeControl. It was created in less than 5 minutes using Rational Rose v4.0.14. This type of diagramming speed is available once all of the Codebook classes are reverse engineered and defined in a Rational Rose Model. Notice what it tells you at a glance.

1. A SAVIContainter IS-A cContainer
2. A SAVIControlContainer IS-A SAVIContainer and it HAS-A SAVILabel aggregated with it
3. A SAVIContainedComboBox IS-A SAVIControlContainer and it HAS-A SAVIComboBox aggregated with it.
4. A SAVIContainedQBDateRangeControl IS-A SAVIContainedComboBox that HAS two SAVILabels and two SAVIDateText boxes aggregated with it.
5. The control gets its first label in the SAVIControlContainer class. This is logical since all contained controls have a label.
6. The ComboBox is added in the SAVIContainedComboBox class.
7. Two additional date text boxes and their corresponding labels are added in the SAVIContainedQBDateRangeControl Class



SAVI Codebook Application Form

I would like to receive a ***FREE*** and ***UNLIMITED*** copy of the SAVI. Stapled to this sheet you will find a page from the original Codebook book that proves I own a legitimate copy of Yair Alan Griver's Codebook (original price approx. $40.00) and a business card (if available). I understand that the information displayed in **BOLDFACE** is mandatory and I will not receive my package if those portions of the application are left incomplete.

First Name _____

Middle Initial _____

Last Name _____

Company Name _____

Address _____

_____

_____

City, State, Zip Code _____, _____ _____ -
_____

e-mail Address *
_____

Work Phone Number _____

Work Fax Number _____

Home Phone Number _____

I would like the SAVI Codebook Framework for _____ VFP Version 5.0a
_____ VFP Version 3.0b

_____ I would like to receive the Rational Rose (V4.0.14) model for Codebook as well,
VFP 5.0a version only.

Send completed application form to:

To: Software Assets of Virginia, Inc.
2109 Silbert Road
Kent Park
Norfolk, VA 23509-2126 USA
Attention: Free Codebook Offer

** - e-mail is the only method of delivery. If you do not provide a ***legible*** e-mail address, you will not receive the framework.