Applying Custom Objects to Existing Code

By Brian Hipple

Experienced BBx® developers, as well as developers with a background in object-oriented design and object-oriented programming, may be a little skeptical when they hear that the latest generation of the BBx language is now 'fully object-oriented' and therefore may have many questions about BBj® custom objects.

This article addresses such questions as "Do BBj custom objects support inheritance, interfaces, encapsulation, polymorphism, access modifiers, and scoping as other object-oriented (OO) languages like Java, C++, or .NET do?" "Is the syntax more comparable to traditional BBx or to these other OO languages?" "What are the benefits of incorporating custom objects?" "What is involved in moving from a procedural coding style to an object oriented style?" And, last of all, "How would a developer integrate custom objects to an existing application or system?"

Object-Oriented Functionality Support

BBj does indeed support traditional OO functionality from inheritance to scoping, giving the application developer the most powerful and versatile development infrastructure.

Picking up the custom object syntax is quite easy for both seasoned BBx developers and experienced OO programmers, as it is a nice blend of BBx and OO type code. Similar to how **DEF FN and FNEND** mark the beginning and ending of a function in BBx, the **method** and **methodend** verbs mark the beginning and ending of a method. Traditional OO statements such as **`class`**, **`extends`**, **`interface`**, **`implements`**; and **`private`**, **`protected`**, and **`public`** access modifiers help make the move from other OO languages into BBx an easy transition.

Benefits of Custom Objects

Benefits of incorporating custom objects into an application or system include decreased development time and increased development resources, readability, maintainability, and reusability. Since many colleges and universities teach OO design and OO programming, this is a new widespread opportunity for the BASIS community to tap into the wealth of generally available programming resources. Development time decreases significantly with the new ability to complete code in the BASIS IDE. Code completion provides developers with the ability to view and select methods or objects from a pop-up list. Developers no longer need to open and inspect a source file to determine what functions or routines to call – they just simply select the desired object or method from a popup list. Readability and maintainability of code increases as all related data and functions used to modify this data are contained in custom objects as human readable attributes and methods.

Reusability of code is the major benefit of using custom objects. Developers can write and debug functionality once and reuse it repeatedly via inheritance, encapsulation, and interfaces. For example, a BBj application that uses custom objects will need access to the BBjAPI and should provide the ability to process application events. Creating a BBj Graphical User Interface (GUI) application using custom objects requires access to the BBjAPI as well as access to the BBj SysGui API and the application resource. The classes that represent this functionality appear in **Figure 1**.

```
rem BBjApp class definition
class public BBjApp
    field protected BBjAPI API!
rem Default Constructor
    method public BBjApp()
        rem Initialize member variables
        #setAPI(BBjAPI())
    methodend
rem Process events for the BBj application
    method public void processEvents()
        process_events
    methodend
classend

rem BBjGUIApp class definition
class public BBjGUIApp extends BBjApp
    field protected BBjNumber SysGuiChan
    field protected BBjSysGui SysGui!
    field protected BBjString Resource$
rem Default Constructor
    method public BBjGUIApp()
        #super!(); rem Call the super
        rem Initialize member variables
        #setSysGuiChan(unt)
        open(#SysGuiChan)"X0"
        #setSysGui(#getAPI().getSysGui())
    methodend
rem Constructor
    method public BBjGUIApp(BBjString p_resource$)
        #this!(); rem Call default constructor
        #setResource(p_resource$)
    methodend
classend
```

Figure 1. Classes that represent the base for a BBj GUI application

The **BBjApp** class has as an attribute, a BBjAPI object (**API!**), which provides access to the BBj API and the method **processEvents** that handles user events for the application. The **BBjGUIApp** class extends the **BBjApp** class to get access to the BBj API and process events functionality, but also has as attributes such as the channel that the sysgui is opened on (**SysGuiChan**), access to the SysGui API (**SysGui!**), and the application resource (**Resource$**). Once the developer writes and debugs these classes, other classes can use these classes and benefit from this proven functionality.

continued...

Brian Hipple
QA Test Engineer Supervisor

Procedural to Object-Oriented

Migrating from a procedural coding style to an OO style involves approaching application development and implementation with a different mindset. Normally, in a procedural application design and implementation, action is performed in a top-down fashion and then refined by adding more details. The task breaks down major functionality into parts comprised of a set of ordered actions. There is usually no relationship, or a very loose one at best, between data and the manipulation of the data. By contrast, in an OO application design and its implementation, action takes place in bottom-up fashion, where individual parts of the task are accomplished in detail and parts are then linked together to form the application. In addition, in OO programming a class is comprised of data called *attributes;* and *behaviors*, also referred to as methods. Being in the same class provides the strongest relationship between data and the manipulation of the data. To understand the references in this article to classes and objects more conceptually, think of "class" as a cookie cutter and the "object" as a cookie.

Migration to Custom Objects

As mentioned earlier, to incorporate custom objects into an existing application, start from the bottom-up. Identify what data the application uses – not just the data on disk, but data structures and groups of related information used in the application. Uniform Modeling Language (UML) object diagrams will now represent custom objects that encompass the application or system. These diagrams are a real help in designing and documenting the application as they describe the custom objects attributes and functionality as well as showing the relationships between the objects. **Figure 2** illustrates design classes with UML diagrams that contain these attributes.

If an application accesses customer information that includes the customer's name, address, phone number, etc., then this data can be added as attributes of a class (**Customer**). Next, determine what action to perform on this data. To be able to set and get this customer information, this class should include methods called accessors that perform these functions (i.e. **setCustomerName()**, **getCustomerName**()). Custom objects provide default accessors for each class attribute. If the custom object has a **CustomerName** attribute, the class will get, by default, a **setCustomerName** and **getCustomerName** accessors with the same access as the attribute.

The next step is to determine what access is required for these class attributes. Is this data only necessary within this class, a sub-class, or is this information for all? This answer determines the type of access for these attributes and methods: private, protected, or public. Aggregation will be necessary if there will be multiple instances of your data. In this case, there will be more than one customer and so there is a need for a class to manage this information. This class (**CustomerMgr**) will contain all customers in some sort of a collection and provide access to remove, retrieve, and edit a particular customer. Keep in mind that when performing OOD, the developer should separate or loosely couple classes used for application data, user interface (UI), and communication. This is important when you want to replace or manipulate one of these areas without affecting the other. To illustrate this, our sample has a class (**CustomerMgrUI**) that implements the necessary UI functionality for the user to add/remove/edit customer information. However, this class does not directly manipulate the data that contains customer information - it contains the class (**CustomerMgr**) which manipulates the data. Therefore, changing the UI from a CUI to a GUI only changes the UI class (**CustomerMgrUI**) and does not affect the data or access to the data.

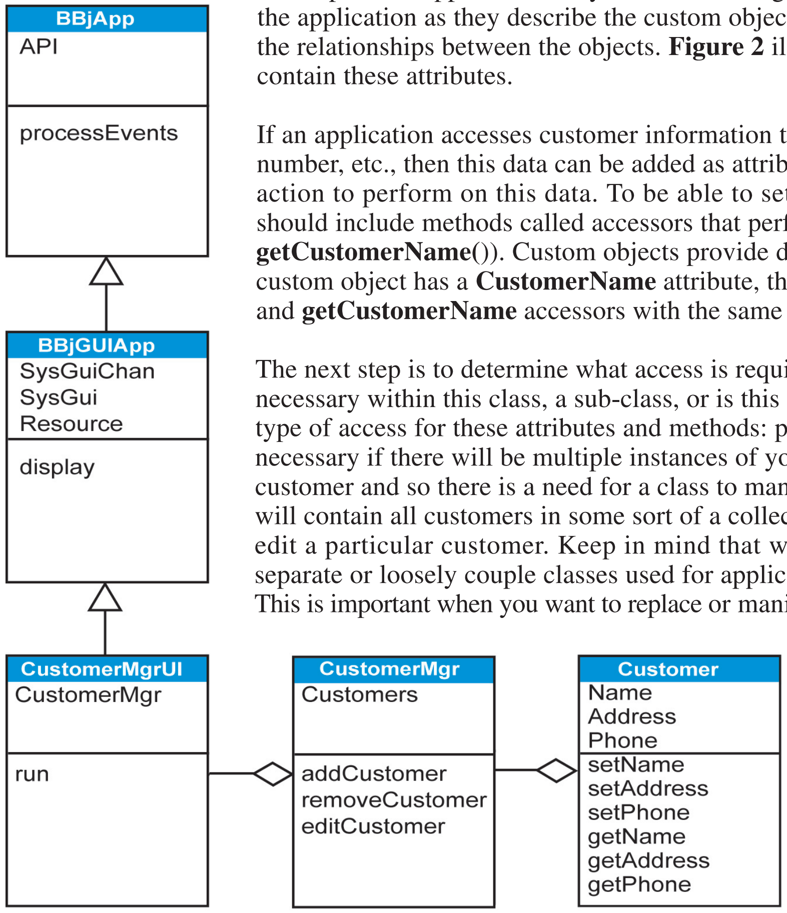

Figure 2. Uniform Modeling Language (UML) object diagram illustrates design class attributes

Summary

Remember, you do not have to move your whole system to custom objects all at once, you can migrate a piece at a time. BASIS strongly encourages the use of custom objects for your next development project. Developers new to OOD and OOP can find a wealth of books and articles to help in this endeavor. Use UML object diagrams and plan to spend at least 30% of your development time in design. To rework existing code or write new code, take a little time to figure out what your data is, determine how the data is manipulated, and what access is necessary to the data. Following these steps will make your next BBx project using custom objects a successful one!

For more information, read the articles *A Primer for Using BBj Custom Objects* on page 11 in this issue and the e-article *Freedom of Choice: Using Object Code Completion in the IDE* located online at www.basis.com/advantage/mag-v10n1/codecompletion.html