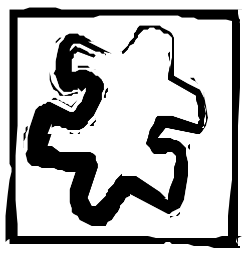
Business Object JumpStart 1.03

Developing .NET Enterprise Applications, Faster



John Kanalakis
solutions@johnkanalakis.com

June 30, 2003

Table of Contents

Creating a Successful Software Project 3
Using Commercial Code Generators 3
Creating Business Objects 3
Planning Your Development Project 4
Outlining Your Business Requirements 4
Mocking-Up Your Application's User Interface 4
Specifying Your Business Objects 5
Defining Your Database Schema 5
Using Business Object JumpStart 5
Creating a .NET New Project 5
Defining an Application Business Object 5
Configuring the Database Connection 7
Reviewing the Files Created 7
Taking a Closer Look at the Business Object 8
Taking a Closer Look at the Business Object Collection 12
Taking a Closer Look at the Stored Procedures 13
Interacting with the Business Object Manager 16
Summary 18

Creating a Successful Software Project

Application development is usually faced with two critical factors: time-to-market and ever-increasing quality. Time-to-market is critical in that any project that an individual or company may have in mind is probably already in the works elsewhere. When that is the case, everything comes down to being the first to deliver… even if it falls short. A Gartner Research study showed that competing products that offered similar features will take the most market share if released earlier. The study further added that the product released first, even with fewer features, typically builds market share faster. The lesson of that study is that it is important to release a 1.0 version of a new product concept as quickly as possibly, then follow-up with feature add-ons over time.

Product consistency and quality are also critical to the success of new products making their debut. Applications with modules that look differently can undermine the application user's confidence in the product. The lower their confidence in the application, the less they use the product and come to depend upon it. The same can be said at the code level. The more modules that are implemented consistently, the easier different developers can step-in to investigate problems.

Using Commercial Code Generators

Some caution needs to come with using commercial code generators. Many solutions take initial input parameters, then build-out an initial code base. In fact, Visual Studio .NET does much the same with their built-in templates and wizards. The solutions to look out for are those which place too much dependency upon the code generator. For example, if a solution provides you with all of your startup code, but that code is not extendable or depends upon a DLL that can not be debugged, then that solution may cause more problems than it solves.

The Business Object JumpStart factors all of this into its offering. The JumpStart was built based upon internal need to produce software quickly by automating the most mundane aspects of building an application. As its name suggests, the Jumpstart focuses on building business objects.

Creating Business Objects

Business objects are defined as memory structures that abstract application entities from their persisted state. In easier terms, business objects hold application-specific data structures without knowing how the data is saved or how the data is presented on screen. This middle-tier approach to application development ensures tremendous flexibility during development. Since there are no pre-determined limitations on how the data is saved, an application can switch database vendors relatively easily. And since there are no assumptions about the application user interface, business objects can easily used by desktop applications, Web applications, and even mobile applications.

The JumpStart takes on the task of creating the source code that represents custom application business objects. The JumpStart also go through the effort of creating stored procedures that save the business objects to a database and to later retrieve them. Coding both of these elements from scratch is not an overwhelmingly complex task, but it does take time and it can be done inconsistently. The JumpStart simply saves you the time it takes two create these elements and does so in a consistent manner.

Planning Your Development Project

Developing an application should be broken into the following steps: outlining the business requirements, mocking-up the user interface, specifying the business objects, defining the database schema, and code implementation.

Outlining Your Business Requirements

The first step of application development must be to outline the business requirements. The produced document has taken on a number of different names by different development groups, including *Functional Specification* and *Requirements Document*. Regardless of the name, a document needs to exist that describes what the application will do and how it will do it. The focus needs to be on pure functionality, not appearance.

This document is important because it informs all project members of what the application will do. There should be enough detail within the document to visualize how the application will flow from start to finish.

Mocking-Up Your Application's User Interface

The next step is to create a mock-up of the application user interface. This step helps to solidify the application features and present a picture of what the application user will be working with. Development teams that can afford it should invest in usability testing. Watching how users interact with a prototype will go a long ways towards a product's success.

Development teams should also include customers as much as possible during the application design phase. If your development team has no customers, find some. Identify your top three or four target customers and hit them up for a "pilot program". The benefit to them is usually free software, or better, software that will meet their specific needs. The benefit to you is great feedback and a very strong lead that will likely convert to a customer, or a reference that can be named when selling to other customers.

In any case, the application user interface needs to be drawn out and documented. You can either build a featureless prototype to take screen-captures, or hand-draw user interface views, scan them in, and paste into a document. Either way, the document will also need to include a description of the view and include a table of user interface controls that appear within the view.

Specifying Your Business Objects

After the business requirements are outlined and the user interface has been drawn, the next step is to specify the application's business objects. Business objects are logical data structures that abstract application entities.

Defining Your Database Schema

The business objects help you to identify what data structures your applications will need to interact with in memory. The next step in application development is to create the database schema that will store the values that fill those business objects. In many cases, much of the database schema will mirror the structure of the business objects. Additional tables will include various lookup tables that the application will need. Also, the database schema will need to map out specific relationships between the various tables as well as constraints on the data values stored.

Using Business Object JumpStart

The Business Object JumpStart is an add-in that seamlessly integrates with the Visual Studio .NET environment.

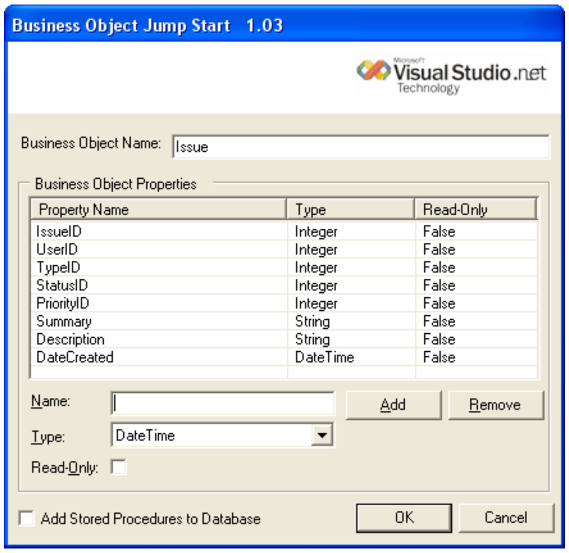
Creating a .NET New Project

Begin by creating a new project or opening an existing one. Although the business objects will be generated in the C# language, the mixed-language support provided by Visual Studio will allow you to work with other languages, such as Visual Basic. In this case, your application code can be completely in Visual Basic and interact with the business objects created in C#.

Defining an Application Business Object

To add a new business object to your application, open your solution within the Visual Studio .NET environment. Next, select Tools > Business Object Jumpstart from the menu. When the JumpStart's user interface is displayed, enter the new business object's attributes as illustrated in Figure 1.

The business object attributes in Figure 1 match a pre-determined object definition summarized in Table 1. Begin by specifying the business object name, in this case **Issue**. Next, enter the attributes one at a time by specifying its name, data type, and read-only status. Read only attributes will be populated when the object loads. Read-only values can be retrieved by other application code, but not set. To have the database select, insert, delete, and update stored procedures automatically created within the database, click the **Add Stored Procedures to Database** checkbox.


Business Object Jump Start 1.03
Visual Studio .net Technology

Business Object Name: Issue

Business Object Properties

Property Name	Type	Read-Only
IssueID	Integer	False
UserID	Integer	False
TypeID	Integer	False
StatusID	Integer	False
PriorityID	Integer	False
Summary	String	False
Description	String	False
DateCreated	DateTime	False

Name:

Type: DateTime

Read-Only: ☐

☐ Add Stored Procedures to Database

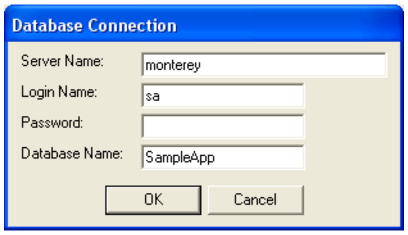
Figure 1. Enteri ng the business object attributes

Table 1. Summary of the Business Object Attributes

Attribute	**Data Type**	**Description**
IssueID	Integer	The primary identifier for a specific issue
UserID	Integer	The user identifier associated with this issue
TypeID	Integer	The type identifier associated with this issue
StatusID	Integer	The status identifier associated with this issue
PriorityID	Integer	The priority identifier associated with this issue
Summary	String	A simple description for this issue
Description	String	The complete description for this issue
DateCreated	DateTime	The date and time this issue was created

Configuring the Database Connection

If you have chosen to have the stored procedures automatically created within your SQL Server database, then you will be prompted to enter the database connection information after you click the OK button. Figure 2 illustrates the Database Connections dialog box. Enter the name of your server, your administrative login name, password, and target database name.



Server Name: monterey
Login Name: sa
Password:
Database Name: SampleApp

Figure 2. Entering database connection information

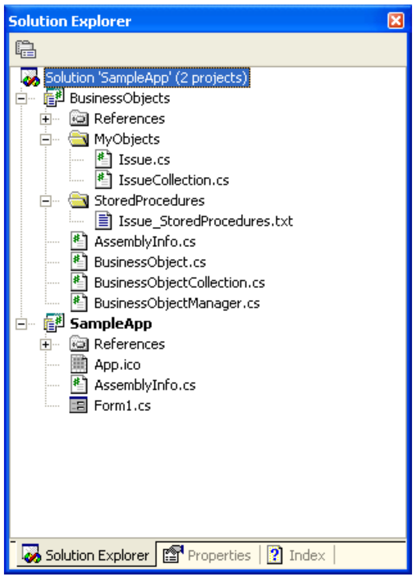
To help streamline the development of multiple business objects, the database connection settings are remembered from the last time it was set. The JumpStart will use this connection information to open a connection with your SQL Server and create the related stored procedures.

Reviewing the Files Created

After you have specified a new business object, have entered it attributes, and have set the database connection settings, the JumpStart takes care of the rest. A new C# class library project is added to your solution, named “BusinessObjects”. Again, although this is a C# project, all business object code is accessible by applications coded in other .NET supported languages, including Visual Basic.Figure 3 illustrates the BusinessObjects project and its members added to the Solution Explorer.

The project includes three files that support the business object framework: BusinessObject.cs, BusinessObjectCollection.cs, and BusinessObjectManager.cs. The first two file implement the base classes for a business object and a business object collection. The last file implements the engine that interacts with the database to read business object values from the database or to write business object values to the database. If you specified the database connection values during the business object creation, then the BusinessObjectManager will default to a customized connection string using the same properties. In this case, you can immediately start using the BusinessObject manager to start storing and retrieving values.

The BusinessObjectManager uses .NET’s object reflection capabilities to study the internals of a business object. These internal values are dynamically mapped to stored procedure parameters for data storage or retrieval.



```
Solution 'SampleApp' (2 projects)
  BusinessObjects
    References
    MyObjects
      Issue.cs
      IssueCollection.cs
    StoredProcedures
      Issue_StoredProcedures.txt
    AssemblyInfo.cs
    BusinessObject.cs
    BusinessObjectCollection.cs
    BusinessObjectManager.cs
  SampleApp
    References
    App.ico
    AssemblyInfo.cs
    Form1.cs
```

Figure 3. A look at the created files within the Solution Explorer

In addition to the three business object framework files, two additional folders are added to the BusinessObjects project: MyObjects and StoredProcedures. The MyObjects folders are filled with dynamically generated business object source files. Each source file is named after the business object that it implements (for example: Issue.cs implements the Issue business object). Also, for each business object, two source files are created, one that implements a single business object and another that implements a collection container for multiple business objects of that type.

For each business object created, a text file summarizing the stored procedures associated with that business object is created and written to the "StoredProcedures" folder. Each text file includes create statements for the Select, SelectAll, Insert, Update, and Delete stored procedures. If have chosen to not auto create the stored procedures within your database, you can paste these create scripts into the SQL Query Analyzer and execute them. The end result will be the same, new stored procedures that are invoked by the BusinessObjectManager to fill and persist your business objects.

Taking a Closer Look at the Business Object

The dynamically createdbusinessobject inherits from theBusinessObject base class. All attributes are specified as private and have matching "get" and "set" accessor methods. Since the business object source code is provided and is editable, you can add your own custom functionality to the business object implementation. Listing 1 presents the "Issue.cs" source file created to abstract an issue. Later, you will see how this object is used within your own application.

Listing 1. A dynamically created Issue Business Object

```
using System;

namespace BusinessObjects
{
    public class Issue : BusinessObject
    {
        private int _IssueID;
        private int _UserID;
        private int _TypeID;
        private int _StatusID;
        private int _PriorityID;
        private string _Summary;
        private string _Description;
        private DateTime _DateCreated;

        public Issue()
        {
            return;
        }

        public int IssueID
        {
            set
            {
                _IssueID = value;
            }
            get
            {
                return _IssueID;
            }
        }

        public int UserID
        {
            set
            {
                _UserID = value;
            }
            get
            {
                return _UserID;
            }
        }
```

```
public int TypeID
{
    set
    {
        _TypeID = value;
    }
    get
    {
        return _TypeID;
    }
}

public int StatusID
{
    set
    {
        _StatusID = value;
    }
    get
    {
        return _StatusID;
    }
}

public int PriorityID
{
    set
    {
        _PriorityID = value;
    }
    get
    {
        return _PriorityID;
    }
}

public string Summary
{
    set
    {
        _Summary = value;
    }
    get
    {
```

```
                return _Summary;
            }
        }

        public string Description
        {
            set
            {
                _Description = value;
            }
            get
            {
                return _Description;
            }
        }

        public DateTime DateCreated
        {
            set
            {
                _DateCreated = value;
            }
            get
            {
                return _DateCreated;
            }
        }

    }
}
```

Taking a Closer Look at the Business Object Collection

The business object collection created to manage multiple Issue objects is also dynamically created. The collection is ultimately managed by an ArrayList collection, but adds a typed interface to streamline code. Most of the collection code is implemented within the BusinessObjectCollection base class. Listing 2 presents the dynamically created IssueCollection class.

Listing 2. A dynamically created Issue Business Object Collection

```
using System;

namespace BusinessObjects
{

      public class IssueCollection : BusinessObjectCollection
      {
              public IssueCollection()
              {
                      return;
              }


              //create a new instace of the Issue object
              public override BusinessObject New()
              {
                      return new Issue();
              }


              public void Add( BusinessObject argObject )
              {
                      _Objects.Add( argObject );
                      return;
              }

      }
}
```

Taking a Closer Look at the Stored Procedures

The stored procedures that are dynamically generated are target for the SQL Server (2000 and higher) database engine. When the stored procedures are created automatically, they can be immediately seen by opening the SQL Server Enterprise Manager and navigating to the target database as illustrated in Figure 4.

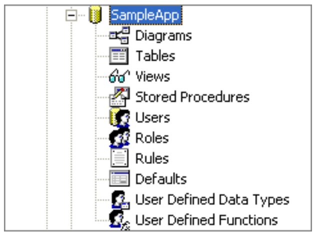
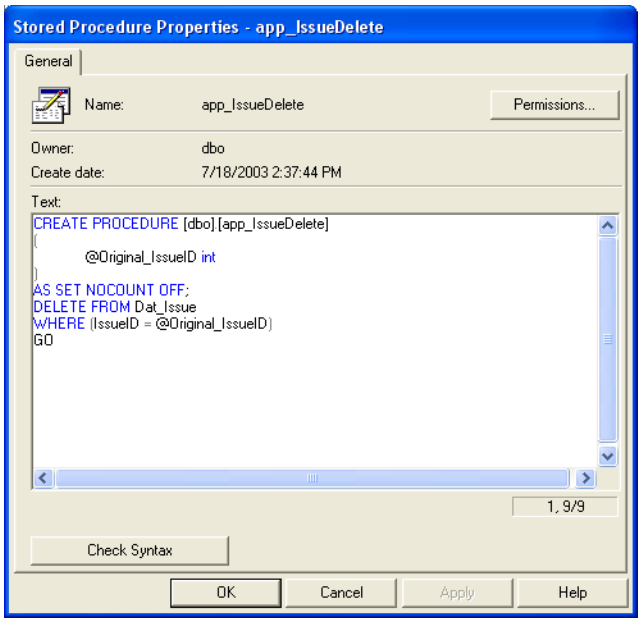


Figure 4. Viewing the stored procedures in the SQL Server Enterprise Manager

To create the stored procedures manually, navigate to the Stored Procedures node within the Enterpise Manager and right-click "New Stored Procedure…" from the context menu. Then, copy & paste the stored procedure create script that was generated by the JumpStart as illustrated in Figure 5.


Stored Procedure Properties - app_IssueDelete

Name: app_IssueDelete

Owner: dbo

Create date: 7/18/2003 2:37:44 PM

Text:

```
CREATE PROCEDURE [dbo].[app_IssueDelete]
(
        @Original_IssueID int
)
AS SET NOCOUNT OFF;
DELETE FROM Dat_Issue
WHERE (IssueID = @Original_IssueID)
GO
```

Figure 5. Manually adding the stored procedures in SQL Server Enterprise Manager

The stored procedures that are dynamically generated are tailored to match the business object that you specify. For each business object, five stored procedures are created: SelectAll, Select, Insert, Update, and Delete. Each stored procedure carefully obeys a specific naming convention as follows:

```
app_(objectname)(procedure name)
```

The “app_” prefix indicates that the stored procedure is application-specific. The object name that follows is populated with the business object name that you have selected. The procedure name is one of the five stored procedures that are required. Listing 3 presents the dynamically generated stored procedure file associated with the Issue object specified earlier.

Listing 3. The Issue related stored procedures

```
CREATE PROCEDURE [dbo].[app_IssueSelectAll]
AS SET NOCOUNT ON;
SELECT IssueID, UserID, TypeID, StatusID, PriorityID, Summary, Description,
DateCreated
FROM Dat_Issue
GO

CREATE PROCEDURE [dbo].[app_IssueSelect]
(
      @IssueID int
)
AS SET NOCOUNT ON;SELECT IssueID, UserID, TypeID, StatusID, PriorityID, Summary,
Description, DateCreated
FROM Dat_Issue
WHERE (IssueID = @IssueID)
GO

CREATE PROCEDURE [dbo].[app_IssueInsert]
(
      @IssueID int,
      @UserID int,
      @TypeID int,
      @StatusID int,
      @PriorityID int,
      @Summary char,
      @Description char,
      @DateCreated DateTime
)
```

```
AS SET NOCOUNT OFF;
INSERT INTO Dat_Issue(IssueID, UserID, TypeID, StatusID, PriorityID, Summary,
Description, DateCreated) VALUES (@IssueID, @UserID, @TypeID, @StatusID,
@PriorityID, @Summary, @Description, @DateCreated)
GO

CREATE PROCEDURE [dbo].[app_IssueUpdate]
(
      @IssueID int,
      @UserID int,
      @TypeID int,
      @StatusID int,
      @PriorityID int,
      @Summary char,
      @Description char,
      @DateCreated DateTime,
      @Original_IssueID int
)
AS SET NOCOUNT OFF;
UPDATE Dat_Issue SET
IssueID = @IssueID, UserID = @UserID, TypeID = @TypeID, StatusID = @StatusID,
PriorityID = @PriorityID, Summary = @Summary, Description = @Description,
DateCreated = @DateCreated
WHERE (@IssueID = @Original_IssueID)
IF @@ROWCOUNT=0        RAISERROR ('Warning:Optimistic concurrency failed.', 10,
1)
GO

CREATE PROCEDURE [dbo].[app_IssueDelete]
(
      @Original_IssueID int
)
AS SET NOCOUNT OFF;
DELETE FROM Dat_Issue
WHERE (IssueID = @Original_IssueID)
GO
```

These stored procedures are designed specifically to interact with the provided BusinessObjectManager class. However, you can certainly customize the procedure to meet your specific needs as long as the stored procedure name is not changed and that the parameter names match the coded business object names. These stored procedures can also be accessed independently from ADO, without using the BusinessObjectManager.

Interacting with the Business Object Manager

Now that you have created your own application and a few custom business objects with the help of the Business Object JumpStart, its time to link the two together. Figure 6 illustrates a sample application with two views: a list of all the issues stored in the database at the top, and a detailed look at a specific issue at the bottom.

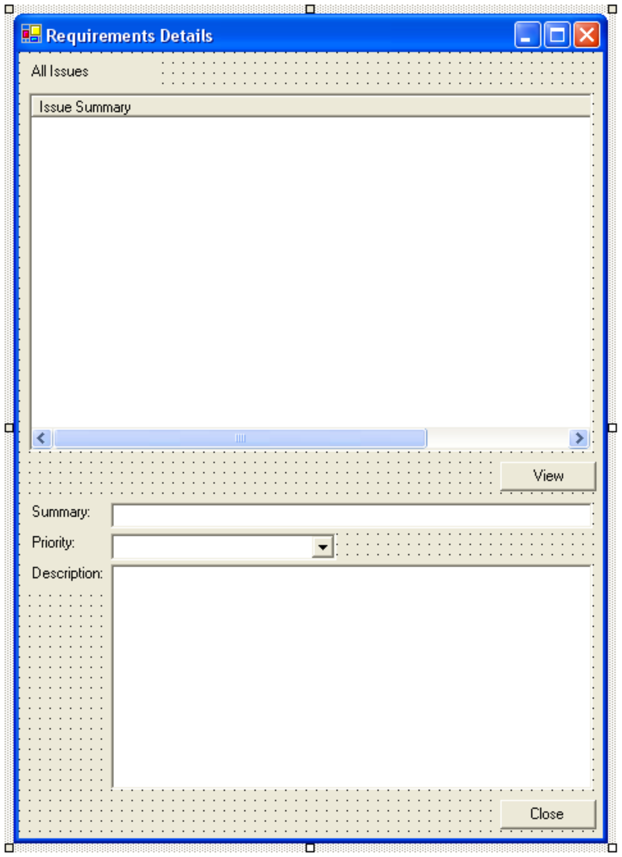


Figure 6. A sample application that presents multiple issues and a specific issue

When the sample application first loads, all issues are retrieved from the database and listed within theListView control at the top. Listing 4 implements the form's Load event handler responsible for this task.

First, an instance of the BusinessObjectManager is created. If you specified a database connection, then the ConnectionString attribute does not need to be set as it has already has been defaulted to point to your database. Otherwise, specify the connection string explicitly using the ConnectionString attribute. Next, create an instance of the IssueCollection class and pass it as a parameter to the BusinessObjectManager's SelectAll method. This method will fill the collection with multiple business objects. After that, simply iterate through the collection to interact with the specific business objects. In this example, new ListViewItem objects area created, containing an IssueID value and the summary string.

Listing 4. Listing all issues within a collection inside the application's load event handler

```
private void Form_Load(object sender, System.EventArgs e)
{
    BusinessObjectManager mgr = new BusinessObjectManager();

    mgr.ConnectionString = "workstation id=MONTEREY;packet size=4096;user " +
        "id=sa;data source=MONTEREY;persist security info=False;initial " +
        "catalog=SampleApp";

    IssueCollection collIssues = new IssueCollection();

    if( mgr.SelectAll( collIssues ) )
    {
        foreach( Issue itemIssue in collIssues )
        {
            ListViewItem itemEntry = new ListViewItem(
                new string[]{itemIssue.Summary, itemIssue.IssueID.ToString()} );

            lstIssues.Items.Add(  itemEntry );
        }
    }

    return;
}
```

In addition to presenting a list of issues, your application can also display any specific issue. Listing 5 implements an event handler that responds to a double-click event associated with the ListView control. Anytime that an entry is double-clicked, the business object's details are loaded and displayed in a set of user interface controls at the bottom of the form.

Listing 5. Displaying a single issue object's values

```
private void lstIssues_DoubleClick(object sender, System.EventArgs e)
{
    Issue selectedIssue = new Issue();

    mgr.SelectOne( selectedIssue,
        int.Parse( lstIssues.SelectedItems[0].SubItems[1].Text ) );

    txtSummary.Text = selectedIssue.Summary;
    cboPriority.Text = selectedIssue.Priority.ToString();
    txtDescription.Text = selectedIssue.Description;

   return;
}
```

The event handler begins be creating a new Issue object. The BusinessObjectManager's SelectOne method is called, passing the new Issue object along with an identifier that indicates which record to retrieve. The end result is a populated Issue object, whose attributes can be assigned to the various user interface controls.

Summary

In summary, the Business Object JumpStart is not meant to do everything for every business application. It is intended to bring structure to an application and to minimize the amount of tedious coding that a structure imposes.

I have found the JumpStart to be a very useful tool for quickly building new business applications. Before long, your business objects will start proving their value as they are re-used by different applications or within different presentation layers. This helps keep your application flexible, especially if considering a port between a Web application, a desktop application, and a mobile application. All presentation mediums are capable of interacting with the same business objects, saving you even more development time.