Business Object Primer

Overview:

Visible Developer is an intuitive enterprise software design and code generation tool that is an add-in to both Visual Basic 6.0 and Visual Studio .NET. It generates 3-tier business applications in a layered architecture directly inside the IDE (integrated development environment). Each layer within the generated application follows a well-defined structure implemented by the template-like “code patterns”.

Information about the physical database - tables, fields and relationships, is imported into Visible Developer’s schema and stored internally in a repository. (There is no need to maintain a connection to your database during the design process.) Visible Developer also imports the schema information from Visible Analyst, Visible’s enterprise modeling tool. Visible Developer’s repository is portable and can be shared by several simultaneous users.

After the import process has been completed the imported schema is further refined. Look-up tables, domains and user-defined relationships are created. (Visible Developer 4.0 allows you to relate tables that physically reside on 2 different physical databases. The data coming from the 2 databases can now be displayed by a single business object).

The imported schema becomes the basis for Visible Developer’s business objects. At the core of Visible Developer’s model are the business objects comprised of properties, methods, rules and views. Additional design-time attributes such as the layouts, user-interface control, permitted values, row and column labels, validation criteria, etc. is added in an iterative fashion. User-defined properties are added to store non-persistent business object data.

Some of the objects are hierarchical in Visible Developer’s design model, fields may inherit from a domain and properties inherit from fields. This follows a top-down design approach.

Design information about business objects is language-neutral, and it is translated by the “code patterns” to create the actual implementation. Each code pattern is designed for a specific language, application architecture and implementation. Code patterns generate the user-interface and the logical and physical business objects. One code pattern generates the thin client (web) user-interface while another generates the fat client user-interface. Code patterns generate forms, classes, modules and projects.

There are several benefits of using Visible Developer’s business objects and 3-tier architecture over a conventional software development methodology.

1) **Develop applications rapidly, not recklessly:** The encapsulation of a business object’s definition and implementation enables it to be deployed and used as a self-contained software component. One or more business objects can be quickly packaged and deployed as a single software component.

2) **Scale your business application without rewriting code:** The separation of functionality and responsibilities between the application layers makes the application highly scalable. It is possible for different types of user interfaces to request the services of a business object. The presentation layer (user interface) does not have any idea of how the business object is implemented.

It only knows what the business object can do and how to request those services. Fat clients as well as thin clients use the same logical business object. Business objects have no pre-conceived notions about the presentation styles used and how the data is displayed.

For example, both the ASP.NET user interface and the Windows forms interfaces generated by Visible Developer, use the same business object.

3) **Data Integration:** Visible Developer makes it possible to integrate and combine data from several different database schemas into a single business object. Cross-schema relationships are created by relating tables from 2 different data sources. The business object designer identifies the two related tables and incorporates them into the business object's definition, creating a single integrated business object. The integration process is seamless and Visible Developer handles the plumbing required to pull data from the data sources. In addition, look-up tables created in different Visible Developer schemas could also be used by a single business object.

4) **Make maintenance easier and predictable:** Visible Developer's structured code generation makes software maintenance faster, reliable and efficient. The costs of software maintenance are significantly reduced.

   Obsolete business objects can be easily reengineered and replaced with new ones. Business objects are self-contained entities, so they can be modified without breaking an existing application. There is no need to rewrite the entire application.

5) **Focus on adding business value instead of sweating the technical details:** A 3-tier distributed architecture has many technical details that need to be addressed. Visible Developer implements it for you and allows you to focus on adding business value rather than trying to figure out the technical details.

6) **Financial savings and ROI:** There are significant savings in software development and software maintenance costs. Since Visible Developer generates over 90% of a typical business application, the development costs are greatly reduced. Because the generated code follows a pre-defined structure, software maintenance is predictable and easy which further translates into cost savings over the life of the application.
   http://www.visible.com/Products/Developer/ROI.htm

The following paragraphs describe the 3-tier architecture and the methodologies employed by Visible Developer.

Why 3-tier Architecture?

Introduction

When software visionaries promote the benefits of distributed architectures the term "business object" is frequently used. Somewhere in their discourse any number of the following terms are used in every possible permutation: 3-tier, N-tier and software component. The message, while inspiring, frequently provides a view equivalent to surveying the landscape at an elevation of 30,000 feet. Sorting out clear direction can be a perplexing and frustrating task.

This article is written for software practitioners who, having heard and accepted the message, now yearn for practical guidance on how to transform vision into reality, or more accurately, how to transform vision into code.

No claims for universal truth are made for the content of the following article. Instead, it offers practical guidance, sound principles, and useful examples of how to start building business objects today. The article is based on our experience and the resulting product **Visible Developer** that generates 3-tier business objects for Visual Basic 6.0, .NET and ASP. For more information about **Visible Developer,** please visit our website at http://www.visible.com/Products/Developer.

Each section heading is phrased as a question about business objects. Sections are roughly sequenced from introductory to more advanced. Navigate the sections sequentially or jump into one that strikes you.

Where Are Business Objects In An N-Tier Application?

First, it is easier to say where the business object is not.

- It is not the user interface (UI) that is the top layer in the application.
- It is not the database services or the bottom layer of an application.

Business objects reside "in the middle" below the UI that creates the visual presentation and responds to the user's actions and above the database services layer that manages the physical data structures. Sounds simple enough, but the definition does beg an answer to the question "What is a layer?"

In a layered application design, as shown in Figure 1, communication only occurs between adjacent layers and never skips a layer. For example, the UI communicates directly with the business object and never interacts with the database service layer while the business object, being in the middle, communicates with both the UI and the database services layer.

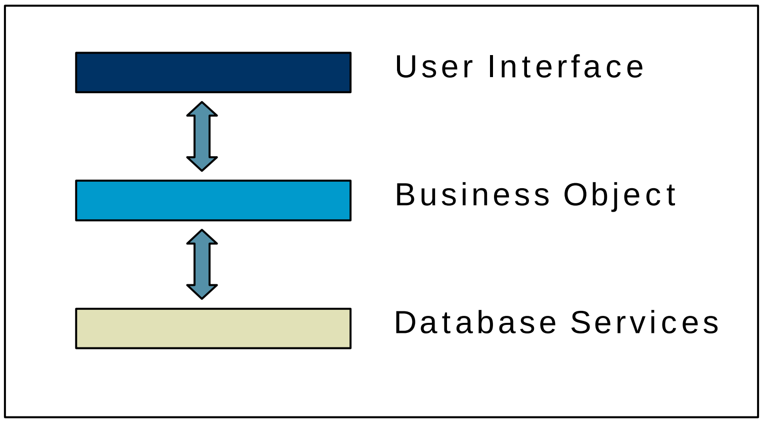

Figure 1. Application Design Layers

Layers within an application design are distinct from the layers or tiers present in the physical deployment of the application. Design layers are software interfaces in the case of **Visible Developer**, that make it possible to distribute the application across different processors creating physical layers. An obvious, but seldom mentioned point is that design layers make distributed architectures possible.

Incorporating layers within an application's design adds complexity and increases the amount of code. So why bother to do it? On the plus side, it provides deployment options enabling an application to scale from single user to enterprise-wide. The following figure shows how adjacent design layers can be packaged together and deployed on a processor.

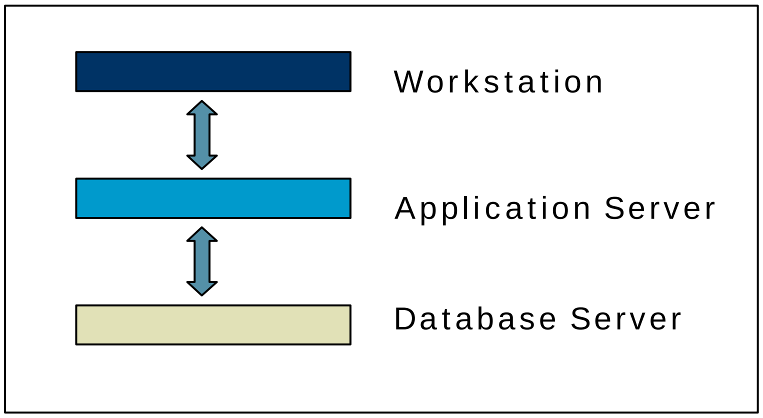

Figure 2. Physical Architecture Layers

In theory, more processors imply greater performance. In practice, however, distributing business applications, and hence business objects, across multiple processors is more involved than implied by the graphics. Business objects designed for a two-tier architecture may perform poorly when moved to a 3-tier architecture.

What causes reality to fall short of theory? The largest impediment is the overhead caused by communication between layers residing on different processors. Techniques such as DCOM and .NET remoting are available to reduce this overhead but they are best discussed in articles dedicated to these topics. The important point to be made is that a business object's design must anticipate communication across process boundaries and utilize one of the available techniques to minimize the performance impact. Later, when code samples are discussed, the techniques used in code generated by **Visible Developer** will be pointed out.

How Do Business Objects Make Reuse Easier?

The short answer is that they don't, at least not simply by virtue of being ActiveX or .NET objects. Good software design promotes reuse and the object-oriented programming model makes it easier to employ good software design practices. The old axiom "garbage in, garbage out" applies; poorly designed business objects will benefit marginally from ActiveX, .NET or any object-oriented technology.

Application of two proven software design principles is a good way to make reuse possible: coupling and cohesion. Coupling is a subjective measure of the degree to which two programs (modules, classes, procedures, etc.) are interdependent. Coupling is unavoidable because if two programs are to accomplish a task together there must be a minimum degree of coupling. The objective is to minimize the amount of coupling in an application.

Cohesion is measure of the "single mindedness" of a component. Components that execute clearly defined tasks each having specified inputs and corresponding results are cohesive. High cohesion is good; low cohesion is bad.

Low coupling and high cohesion make maintenance easier because problems are easier to find and fix because they can be isolated. Reuse is easier because an identifiable unit of reuse is easier to find.

.NET and COM support the object-oriented programming model, and when used appropriately, reduces coupling and makes reuse easier because:

- Public methods define the tasks the business object performs and are visible expressions of the object's cohesion. A poorly constructed set of methods destroys cohesion.
- Properties selectively expose data to the outside world and hide internal information at the same time. When internal data is hidden, coupling is reduced because external objects cannot manipulate or access private data.

An example illustrates how cohesion and coupling appear in a design and how a 3-tier business object improves the design.

The basis of the example is a single database table called Purchase Order with the following fields:

- Number

- Customer Number
- Date_Received
- Status

In the first design a form, frmPODetail, contains a textbox control for each field. A data bound control is added to directly retrieve data from the table. The code behind the form is the only layer in the design. So, what's the problem? Consider the following sequence of events:

> **Requirement 1**: Number is now created by concatenating a three-letter customer code with system-generated number.
>
> **Response**: Code is added to frmPODetail to obtain the customer code and generate the Purchase Order Number.
>
> **Requirement 2:** A request is made for a new form containing a grid control so business users can easily work with multiple purchase orders.
>
> **Response**: A new form, frmPOList is created. The code that was added to frmPODetail to obtain the customer code and generate the Purchase Order Number is also added to this form.

A simple business rule stating how purchase order numbers are formed now appears in the code behind two forms. Maintenance effort is doubled and the probability of a bug is increased. The diagnosis is low cohesion and high coupling. Cohesion is low because the code behind the forms has two very different responsibilities:

- Present data to the user and react to the user's actions.
- Enforce business rules.

The most egregious form of coupling is the data bound control, but more about that later. The solution is to create layers in the software design and separate responsibilities among the layers. The two forms will be in one layer and a second layer will contain a class, PurchaseOrder, responsible for validating and, in the case of Number, generating field values.

Is the problem resolved? Are two design layers enough? Probably not because there is still a high degree of coupling between the data bound controls in the forms and the underlying database structure. Changes to the Purchase Order table would likely require changes to both the grid and detail forms.

A better solution is to remove knowledge of physical data storage (the name of the database, table, columns, etc.) from the UI and move it to a single piece of code. One possible location is the business object created in the prior step. Another possibility is to add another layer responsible for database access. The latter design option is the approach taken by **Visible Developer,** and it is discussed in more detail in later sections.

To summarize, a 3-tier business object design using .NET or COM facilitates reuse by providing an object-oriented programming model. The potential for reuse is achieved only if the business object's design exhibits high cohesion and low coupling. This naturally leads to the next question.

How Do You Design Business Objects?

The art in business object design, and what very few articles or books mention, is how to decide what information and tasks should go into each layer of the design.

The following principles guide the design decisions embodied in code generated by **Visible Developer.**

- Keep the UI simple. A good design principle is to think of the UI as wallpaper; it is the thin visual covering provided for the structural portion of the application. UIs change due to new technology, user requirements and, just like wallpaper, taste.
- Don't allow the business object to make assumptions about the UI. Ideally, business objects should be "UI agnostic" – they don't know of their existence and it doesn't really matter anyway.
- Keep information local to each layer and only share information through properties and methods. The design maxim for each layer is "Everything it needs to know, but nothing else."
- Think of the business object as a provider of services to the UI. It is the responsibility of the UI to make use of these services to create the interaction with the user. A robust UI might take advantage of a business object event to enable the update button on the form. A simpler UI, an Excel spreadsheet for example, might choose to ignore events offered by the business object and limit the interaction to a read-only list of information. If a business object is designed properly, different UIs can reuse its services.
- To be reusable, business objects must expose their essence via the methods, properties, and events offered. Keep in mind that they will be implemented as a dll or exe, so a potential user's knowledge about a business object is limited to the kind of information provided by the object browser. A daunting challenge, isn't it?

Visible Developer applies the above design principles when generating business objects resulting in three distinct design layers plus the database services:

- User Interface
- Logical Business Object
- Physical Business Object
- Database Services (RDBMS)

The User Interface

The visual presentation of information to the business user and responding to user actions is the responsibility of the UI. The only requirement made of the UI is it must be able to create and use .NET or COM objects because business objects are .NET or COM objects. This provides a great deal of flexibility for both application developers and business users. Formal and robust UIs can be written using languages like VB, C++ or .NET. At the same time an expert user can access information contained in business objects through a familiar spreadsheet application.

The Logical Business Object

The logical business object is directly below the UI in the application design. It encapsulates data, business operations and rules governing adding, changing, or removing its data. Logical business objects present information and business operations from the business user's perspective.

Logical business objects do not directly access the underlying database. Instead, they use services provided by the physical business object to read, update, and delete business data.

The Physical Business Object

Physical business objects tend to be much simpler than their logical counterparts. Their responsibility is limited to communicating with the database services layer in response to a request from the logical business object. Physical business objects are aware of the database structure and are responsible for translating the logical business object's request into one or more transactions in the database services layer.

Database Services

The database services layer is typically a relational database management system (RDBMS).

Separating Information and Responsibilities among Design Layers

The leap from concept to implementation requires hard choices and involves many details. Figure 3 describes the design of each layer in a **Visible Developer** application by stating the responsibility of each layer and the information it does or does not possess. Design decisions reflect the principles listed previously with the goals of making each layer cohesive and reducing coupling between layers.

Design Layer	Responsibilities	What It Knows	What It Doesn't Know
1. User Interface	Present business data to the user Respond to actions taken by the user Display error messages	GUI Controls Properties and methods of business objects it uses	Validation rules for properties Business rules Where business data is stored
2. Logical Business Object	Provide services to the UI Enforce business and validation rules Request data services from Physical Business Object	Business rules and validation rules Services provided by the physical business object	The forms and GUI controls displaying its data Where business data is stored
3. Physical Business Object	Respond to requests for data services made by the Logical Business Object Obtain data services from the DBMS	The database containing business information How to request data services from the DBMS	Business rules and validation rules
4. Data Services (RDBMS)	Retrieve data from physical storage Change data in physical storage	SQL and/or ODBC	Everything else!

Figure 4. Overview of Design

The next step is to translate this design into forms and classes.

How Many Classes Are Needed To Create A Business Object?

A layer in the application design does not necessarily equate to a single class**.** It is often useful to implement a layer in the design using more than one class or standard module.

Our concept of a business object grows more complicated. Originally it was described as a software design layer somewhere between the UI and database services. In the previous section, the business object was split into two layers: a logical and physical. And finally, each of these layers contains several components. So a business object is not a single "thing;" in reality, it is a collection of collaborating parts each having a precisely defined role.

Visible Developer generates four classes for each business object; three classes are contained in Layer 2, the logical business object, and one class is contained in Layer 3, the physical business object or persistence layer.

A naming convention is used to easily identify the particular business object class and suggest its purpose. In the following example, the class names appear, as they would be generated for a PurchaseOrder business object.

Classes contained in Layer 2 are:

- PurchaseOrder: The primary object used by the UI. It contains the business object's properties, methods, and business rules. It acts as an intermediary between the UI, Layer 2, and Layer 3. PurchaseOrder uses its counterpart Layer 3 class, PurchaseOrderPersist, to access and change data in the database.
- PurchaseOrderList: A specialized class used when the UI needs a list of information about one or more business objects. To the UI, the PurchaseOrderList behaves like a recordset with each row containing property values of a single business object.
- PurchaseOrderRules(VB only): A specialized class that contains rules and their logic. The rules are contained in the PurchaseOrderList class for .NET.

The Layer 3 class is:

- PurchaseOrderPersist: The Layer 3 counterpart to the PurchaseOrder and PurchaseOrderList classes. It carries out database operations on their behalf.

The distribution of classes among design layers is shown in Figure 5.

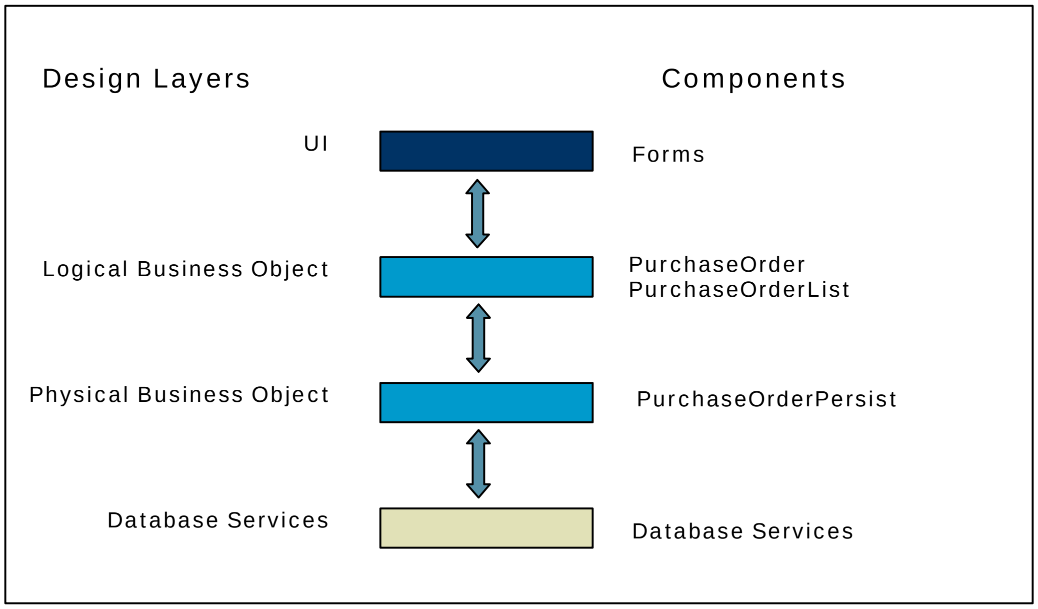

Figure 5. Business Object Components and Design Layers

The final section addresses the obvious question: How does all of this work?

How Do The Classes Interact To Create A Business Object?

Working with Business Objects in Visible Developer

Let's use as an example two tables from a Microsoft Access database: Purchase Order and Customer. Visible Developer automatically extracts information from the database schema and creates a business object from each table. Table fields are included as object properties and four default methods: add, delete, read, and update are created. The relationship between Customer and Purchase Order is also created. **Visible Developer** uses foreign key information when generating code.

Developers can add design information to make the generated code as close as possible to the desired result. For example, developers can select a control type (text box, combo box, option buttons, etc.), list permitted property values, and add more methods.

Visible Developer offers developers a number of code generation options including the ability to generate two styles of forms: List and Detail. Generated components include class modules, standard modules, and forms with controls and code. **Visible Developer** packages generated components in 3 projects depending on the selected generation option. Each project results in a standard executable or a dll.

Layer 1: The User Interface

After code generation is complete, **Visible Developer** is closed and the developer continues working in the IDE. Each layer in the generated application appears as a project.

Components in the first layer are contained in Project_UI. A total of nine forms are generated. frmStartup, is a standard form that is always generated. Each button on the startup form corresponds to a generated business object. Clicking the button displays a list form for that object. In this example, the list forms are frmPurchase_OrderList and frmCustomerList.

List forms display business object property values in a data grid. The number of records returned can be changed by modifying the search criteria on the search form. After a particular customer or purchase order is selected, it can be deleted or changed using a detail form. The detail forms, frmPurchase_OrderDetail and frmCustomerDetail, display property values of a single business object and enable the business user to create, delete, or update using the same form.

Layer 2: The Logical Business Object

The second project, Project_Layer2, contains the class modules that constitute the logical customer and purchase order business objects. **Visible Developer** creates two classes for each business object in Layer 2 of the application: BusinessObject and BusinessObjectList. A brief description of each class and how it is used is provided in a previous section.

Layer 3: The Physical Business Object

Project_Layer3 contains components generated for Layer 3. The classes generated by **Visible Developer** for layer 3 correspond to classes in Layer 2 and the methods for related classes are nearly identical. For example, the Purchase_Order class has an Add method typically invoked by a form as follows:

```
myPurchase_Order.Add
```

Logical classes do not perform database actions. Instead, they rely upon their Layer 3 counterpart, the “persistence classes,” to carry out these tasks. Continuing the example, the Purchase_Order object has the following statement in its Add method:

```
myPurchase_OrderPersist.Add
```

The Purchase_Order class contains business rules and is responsible for knowing if the object is in a correct state to be added. For example, it would contain a business rule, to ensure all mandatory properties have valid values before attempting to add.

The remaining sections use extracts from the generated code to demonstrate the flow of data and methods between the layers.

Sample Code: Searching and Displaying Purchase Orders

The first execution path begins with code in the frmPurchase_OrderList form. The sequence is:

1. Layer 1: List form uses the Search method of Purchase_OrderList to find objects matching the search criteria.
2. Layer 2: Purchase_OrderList object uses the Search method of Purchase_OrderPersist to search the database.
3. Layer 3: Purchase_OrderPersist creates an ADO Recordset containing property values of matching purchase orders and returns it to Layer 2.

Layer 1: Purchase Order List Form: Loading Datagrid

Layer 1 objects, use Layer 2 logical objects to perform tasks. A UI designer uses a list object, like Purchase_OrderList, as a convenient utility object to perform searches and make the results easily accessible. The UI's knowledge of the business object is limited to the names of properties and its only responsibility in this example is to create a string containing expressions of the form **"Number > 100 And Received_Date < 12/12/1998"**. The UI, or form, does not need to know:

- The name of the table or tables providing property values.
- The names of the columns in a table that corresponds to these properties.
- How to make a database connection or even what database (or databases) contains purchase orders.
- That the Purchase_OrderList object will use another object to actually query the database

Moving this knowledge to other layers in the design strengthens the cohesion of the UI and reduces coupling within the application.

The highlighted statement in the above example,

```
myPurchase_OrderList.Search strSearchCriteria
```

leads to the next piece of sample code.

Objects in Layer 2, or logical objects, do not directly access persistent data. As shown in this code sample, the Purchase_OrderList object creates a Purchase_OrderPersist object and uses its Search method to obtain purchase orders matching the criteria.

Purchase_OrderList might appear to be a useless middleman in this transaction but in fact provides valuable services to the UI. It maintains matching purchase orders obtained from Layer 3 internally in a recordset and exposes many of the recordset's methods and properties to the UI. This enables a UI developer to use the Purchase_OrderList object as if it were a recordset; a familiar construct to many programmers.

The Purchase_OrderList object enables the UI developer to avoid the cost associated with creating purchase order business objects for each row by retaining the values internally as a recordset. If a business user requests to work with a particular purchase order displayed in the data grid, the UI developer creates the corresponding Purchase_Order object.

The highlighted statement in the above example,

```
Set rsPurchase_Order = objPersist.Search(strSearch)
```

leads to the next piece of sample code.

The Layer 3 object, Purchase_OrderPersist, understands how to translate between the logical view of the object's properties and the underlying physical view consisting of database tables and columns. This layer is responsible for making the database connection (not shown in the sample code), converting property names to column names (also not shown), creating recordsets. Etc.

After the Search method of the Purchase_OrderPersist object is finished, control passes back to the Purchase_Order object in Layer 2 and from there back to the UI in Layer 1.

Sample Code: Changing A Property Value

The second sample code execution path begins with the frmPurchase_OrderDetail form. The sequence is:

1. Layer 1: The Detail form uses the Read method of Purchase_Order to load a specific purchase order.
2. Layer 1: The Detail form responds to a change to a purchase order number.
3. Layer 2: The Purchase_Order object validates the new number.

A detail form always creates a Purchase_Order object when it is first loaded. The next step depends upon the action taken. If a new purchase order is to be created, the UI displays the form after the Purchase_Order object is created; essentially leaving it in an "empty" state. When updating or deleting a purchase order, the UI uses the Read method to retrieve values from the existing purchase order.

The highlighted statement:

```
myPurchaseOrder.Read strPurchase_OrderObjId
```

loads the current property values of the purchase order identified by strPurchase_OrderObjId from the database. To accomplish this task, the Read method of the Purchase_Order object (Layer 2) creates a Purchase_OrderPersist object (Layer 3) and invokes its Read method. The movement between Layer 2 and Layer 3 is similar to the previous example so it is not shown.

After the Read method of myPurchase_Order object is finished, the UI obtains the value of each property using the RefreshDisplay sub. It contains statements of the form:

```
txtNumber.Text = myPurchase_Order.Number
```

The first step in this example concludes with the form displayed and the business user ready to make changes.

The second step in the sample demonstrates how the UI uses the services of the business object to validate and store property values. In the example the detail form has a textbox, txtNumber, which displays the value of the Purchase_Order's Number property. Whenever the business user tabs through the txtNumber textbox, the UI sets the business object's corresponding property value to the control's value. And that is all it does.

It is important to emphasize what the UI does not do when using business objects:

- It does not check to see if the characters entered represent a number.
- It does not test to see if the business user is permitted to change a value already entered.
- It does not test to see if a Number is mandatory.
- It does not cross check the Number entered with other values.
- And so on….

To sum it up, the UI only reports values to the business object. It does not attempt to interpret them. If a business object discovers an error, it raises an event and it is the UI's job to respond to the event appropriately.

The final lines of code in the sample seem contradictory at first glance. Why does the UI obtain the Number property from myPurchase_Order when it just sent the value to the object in the previous statement? The reason is the strict line of responsibility drawn between the UI and the logical business object. UI's display property values and business objects supply property values. The UI asks the business object for the value of the Number property in case the value originally provided by the UI was reformatted by the business object. Perhaps reformatting purchase order numbers seems unlikely, but consider social security numbers (SSN). Should every control displaying an SSN have code to put the dashes in the right place, or is it a better design to make formatting an SSN the responsibility on a single business object?

The final step in this example shows code from the Property Let in the Purchase_Order business object. The code should look familiar to programmers. It is similar to statements previously contained in a form. The responsibilities, and hence the code, for data validation is removed from the UI and placed with the business object. Encapsulating logic within the business object makes it available for reuse by any UIs, windows forms, web browser, Excel spreadsheet, etc., that uses the business object.

The highlighted code in the sample demonstrates how business rules are incorporated. After a value passes all initial checks, the business object calls all business rules, possibly affected by a change in the property's value. MandatoryValuesExist is called in the sample because Number is a required field and the business rule governing mandatory properties applies to it.