

# **Starting with Bold for Delphi/Bold for C++ Part 1: Introducing the basics**

*By Anthony Richardson  
31 October 2001  
Revision 1.0*

## Contents

<b>OVERVIEW.....</b>	<b>3</b>
Introduction.....	3
About the Author .....	3
Acknowledgments .....	3
Credits .....	3
Part 1: Introducing the basics.....	3
<b>GETTING STARTED .....</b>	<b>4</b>
Introduction.....	4
Example Application.....	4
Requirements .....	4
<b>MODEL BASED DEVELOPMENT .....</b>	<b>5</b>
Introduction.....	5
What is modeling?.....	5
The Example Application Class Model .....	7
<b>BUILDING THE APPLICATION.....</b>	<b>9</b>
Introduction.....	9
Groundwork.....	9
Models, Handles and Persistence .....	9
Basic configuration Changes .....	11
Building the Model.....	12
Adding a User Interface .....	18
<b>RUNNING THE APPLICATION .....</b>	<b>21</b>
Entering Test Data .....	21
Delphi vs. Bold Data Aware Comparison.....	22
Bold Object Editor Forms .....	23
<b>SUMMARY .....</b>	<b>25</b>
Future .....	25

## Overview

### *Introduction*

'Starting with Bold for Delphi/Bold for C++' is a series of articles design to provide an introduction to the Bold products and effective techniques for applying the Bold framework in the development of real world applications.

These articles are designed as an introduction. The Bold for Delphi/Bold for C++ products contains many components and is comprised of over 1000 classes spread across over 350 units. The product contains many sub frameworks within these classes to support advanced development techniques and the application of industry best practices in the form of patterns, interfaces and modeling.

The design of Bold for Delphi/Bold for C++ is carefully layered to enable rapid adoption of core techniques with the gradual adoption of the more complex or sophisticated methods as required. These articles are designed to facilitate the transition from traditional programming to the core Bold for Delphi/Bold for C++ techniques.

### *About the Author*

Anthony Richardson is a software developer based in Adelaide, Australia. Anthony is contactable via email at [anthony@viewpointsa.com](mailto:anthony@viewpointsa.com). More information about Viewpoint (SA) Pty Ltd is available on the Internet at [www.viewpointsa.com](http://www.viewpointsa.com)

### *Acknowledgments*

I would like to acknowledge the assistance of BoldSoft MDE AB in the creation of these articles. Especially the assistance Jesper Hogstrom and Dan Nygren, without their support this project would not have been possible.

### *Credits*

All trademarks are properties of their respective holders. All intellectual property claims are respected. The publication is copyright 2001 by Anthony Richardson. Anthony Richardson grants BoldSoft MDE AB a royalty-free non-exclusive license to distribute this publication worldwide.

### *Part 1: Introducing the basics*

The first installment will address the use of basic Bold for Delphi/Bold for C++ components and an overview of what they are doing behind the scenes. The process of designing and implementing a basic model will be covered; enough 'GUI' work will be implemented to test the design.

## Getting Started

### *Introduction*

This series is going to follow the process of building an application. This will have the goal of showing how Bold for Delphi/Bold for C++ is used to solve the real world problems associated with application development.

The need for expressing concepts with clarity, presenting options and ensuring everyone doesn't fall asleep, means that the example application has to be small. The Bold framework is designed for a range of application sizes and indeed is used on many large government and corporate applications. But the technology is equally at home on small size projects.

### *Example Application*

The application covered by this tutorial will be an address book application, as is often found in popular Personal Information Management (PIM) applications. The reason this was chosen as the example is because:

- A) It's a problem area most developers would be familiar with.
- B) It suits well the goal of explaining various relationship issues, inheritance and user interface issues.
- C) It is equally suited as a desktop application and as a network application.

### *Requirements*

The scenario for this application is a business address book. The basic needs for our application are:

- Store contact information for people.
- Store contact information for companies.
- Allow for people to be associated with a company.

Detailed or even specific requirements for the application are not listed, we will indeed assume these as the application progresses.

## Model Based Development

### *Introduction*

The term model based development is used often when describing the Bold for Delphi/Bold for C++ products. The term refers to the process of developing your application directly from the class model, rather than traditional process of transforming an entity-relationship model into a table structure and then coding your application to the table structure.

The Bold products applies Object Oriented flexibility and design to the persistence layer of your application making design easier. It is worth spending time on this upfront because this is a huge part of the productivity enhancement you will experience when using the Bold architecture.

### *What is modeling?*

A model is a description or view of something. Like a model aircraft is a miniature representation of a real aircraft, a software model is a representation of the domain that the software is designed for. It is possible to generate many different models for a particular software project. Popular software engineering methodologies describe models and activities at various stages of the software development lifecycle which aid in the understanding of the problem domain. Models also simplify communication between developers, users and other stakeholders. By creating the various types of models you can describe a complex software system in a series progressively more detailed set of views into the problem domain.

### *Traditional Entity-Relationship Modeling*

Entity Relationship Modeling (ER Modeling) has traditionally been the method used to describe the business data for an application, the term data modeling is also used. Generally these models are restricted to portraying a static view of the data within a problem domain and fail to recognize the operations and interactions between entities.

After the completion of the ER model a database schema is developed. During this process the types of relationships portrayed in the ER model are transformed into a format that is suitable for the database. Certain types of relationships will result in extra tables being generated and the introduction of various primary and secondary keys. Where data may have been originally design with concepts like inheritance relationships, these models are 'flattened' to allow for the design to fit within the constraints of a relational database. This will tend to result in the loss of information from the model, not in terms of the information stored, but in the implied relationships and business rules of a fully normalized ER model.

### *Class Modeling*

Class Modeling is a modern version of ER modeling that expands on the benefits of diagramming domain entities and their relationships and includes the interaction between those classes. Where ER Modeling is only modeling the data as required to be saved, the class model includes all domain entities

and the operations performed by those entities. This allows for a much more accurate view of the business problem domain. By using class modeling and the various diagramming techniques available it is easier to build the correct software architecture earlier. The classes in a class model may also be referred to as business objects.

### *Business Objects*

The term business object is used in many different ways and means different things to different people. Other terms used for business objects are domain objects and business classes. The process of storing and retrieving these business objects is often referred to as object persistence or object relational mapping. Frameworks that support the use of business objects are often referred to as object persistence frameworks.

Using business objects is the application of object-oriented techniques beyond coding and infrastructure into the realm of the real-world items (objects) being used by your software. An example of a using business objects is part of this article.

### *Unified Modeling Language (UML)*

From the UML website, [www.uml.org](http://www.uml.org) :

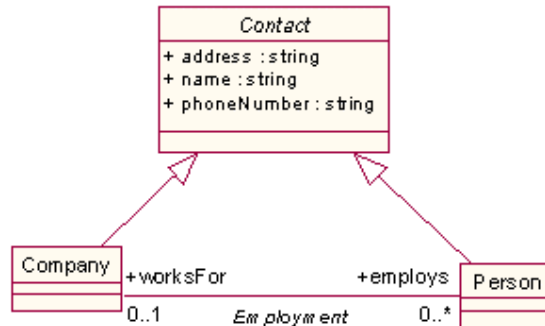
“The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

The UML was developed by Rational Software and its partners. It is the successor to the modeling languages found in the Booch, OOSE/Jacobson, OMT and other methods. Many companies are incorporating the UML as a standard into their development processes and products, which cover disciplines such as business modeling, requirements management, analysis & design, programming and testing.”

UML is critical for the use of Bold for Delphi/Bold for C++. The Bold for Delphi/Bold for C++ product includes it's own UML modeling tool. Learning UML is easy and applying the full range of UML modeling techniques is a rewarding process.

## The Example Application Class Model

In this example program we are going to implement the following UML model for our address book application:



Lets start easy. This design is very basic and ignores requirements for complicated addresses and multiple phone numbers.

If you are not familiar with UML, the diagram shows three classes, represented by the rectangles. The class 'Contact' has its title displayed in italics to indicate it is an abstract class. The lines with the arrowheads indicate that both the 'Company' and 'Person' classes descend from the 'Contact' class.

The line between the 'Company' class and the 'Person' class indicates a relationship. Relationships contain a title, 'Employment', properties for each class, 'worksFor' and 'employs', as well as multiplicity specifiers, the '0..1' and '0..\*'. These will be explained in greater detail soon.

The model shows the following business rules:

- A *Contact* has a name, address and phone number.
- A *Contact* must be either a *Person* or a *Company*.
- A *Company* employs zero or more *Person*.
- A *Person* works for zero or one *Company*.

If this model were implemented in a RDBS more than likely compromises to the design would have to be accepted, with the responsibility for the enforcement of the rules moved to the application logic. For example depending on how the data is 'flattened' the entities *Person* and *Company* may be represented in a single table requiring the relationship type restrictions being implemented in application code.

There is danger when moving from what is essentially an Object Oriented design to a flat relational database design. The Object oriented design clearly portrays real world entities and models valid business rules via relationships and care must be taken to preserve the intentions of the design. Often this can be difficult or impossible, resulting in the need for supplementary specifications. These document the business rules now required to be enforced in code.

In reality this transformation results in an implementation that loses the intent of the original model and may even drop off important business rules if careful documentation processes aren't followed. Indeed even if these rules are documented it is reliant on them being understood and implemented by the application programmer. As with all documentation it must also be manually maintained during the life cycle of the database.

*“This is what  
BoldSoft terms  
the Executable  
Model  
Architecture  
(XMA)”*

The ultimate solution is to use a medium that accurately expresses the data model and is executable in the application. Because the documentation becomes the code, and vice-versa, business rules are maintained and enforced during the entire life cycle of the database application.

This is what BoldSoft terms the Executable Model Architecture (XMA) and is the foundation of the Bold for Delphi/Bold for C++ products.

## Building the Application

### Introduction

It's time to start Delphi and ensure Bold for Delphi/Bold for C++ is installed. The following procedure should be complete enough for anyone that has used Delphi to follow easily. After each series of steps a description of what has been achieved is offered before moving on to the next steps.

### Groundwork

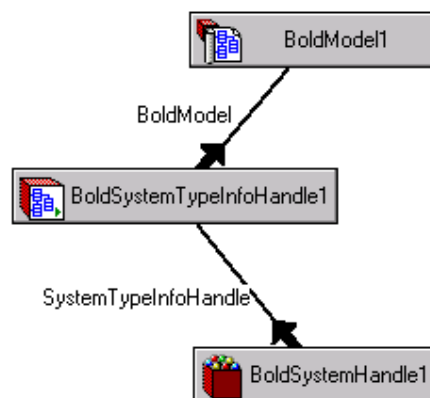
To get started we need to setup a basic Delphi project with a data module.

1. Select *File|New Application* from the Delphi menu.
2. Save the Form unit as *MainForm.pas* and the project as *ContactManager.dpr*
3. Add a new Data Module by selecting *File/New* and selecting *Data Module* from the *New Items Dialog*.
4. Set the Data Modules name to *BusinessModule*.
5. Save the DataModule as *BusinessLayer.pas*.

### Models, Handles and Persistence

The basics for a Bold for Delphi/Bold for C++ application are adding our Model to the application and the chosen persistence layer. To prepare the application to accept a model, do the following:

1. From the *Bold Handles* component tab add a *BoldModel*, *BoldSystemTypeInfoHandle* and *BoldSystemHandle* to the Data Module.
2. Link the *BoldModel* property from *BoldSystemTypeInfoHandle1* to *BoldModel1*.
3. Link the *BoldSystemTypeInfoHandle* property from *BoldSystemHandle1* to *BoldSystemTypeInfoHandle1*.



The *BoldModel* component will hold the model, that is the classes, relationships, constraints and types. Its role at design time is to hold the model and stores it as a string in the Delphi Form/Data module. At runtime it performs an intermediary transformation of the model into a format to be used by the *BoldSystemTypeInfoHandle* and the chosen persistence mechanism.

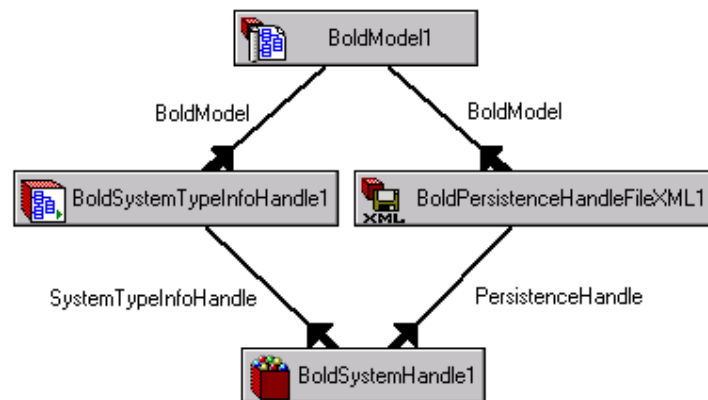
The information stored in the *BoldModel* component can be thought of as metadata, in a similar way that you can query a database to provide table and field information. This component is generally used only at design time except for advanced operations not covered by this article.

The *BoldSystemTypeInfoHandle* component holds the runtime information needed by a *BoldSystemHandle*. The information is an efficient representation of the information in the UML-model, heavily optimized for speed. This component is generally not referenced directly in code, but rather acts as a source of information for other Bold Components.

The *BoldSystemHandle* component is used to represent an entire system of domain elements, also known as an object-space. This component represents the available instances of objects in the model at runtime. This is the primary gateway between design time metadata and actual stored objects.

The components as they stand are fine for a system that doesn't need to save data (persist) between sessions. However few applications have value without saving information between each use. To enable a fast path to running a Bold Application we will add an XML persistence component to the data module

1. From the *Bold Persistence* component tab add a *BoldPersistenceHandleFileXML* component to the data module.
2. Link the *BoldModel* property to the *BoldModel* component.
3. Link the *PersistenceHandle* property of *BoldSystemHandle* to the *BoldPersistenceHandleFileXML* component.



The *BoldPersistenceHandleFileXML* component allows the application to save and read objects from an XML file. This is a very handy component to use while you are still prototyping an architecture as it saves you from having to regenerate your database tables while still making significant changes to your model. Later on we will remove this component and use the BDE or Interbase components to allow use of a relational database. The design of the Bold products allows us to do this without changing our application. This is called database independence. At that time we will also look closer at the SQL calls generated by the Bold framework, but for now we will forget this as it's not critical for using Bold for Delphi/Bold for C++.

## ***Basic configuration Changes***

Initially in this example application we will use the following property values, set these in the Delphi property editor:

Component:        BoldSystemTypeInfoHandle1  
Property:         UseGeneratedCode  
Value:            False

Later on we will enable this and work directly with the model using Object Pascal, but initially this is not required.

Component:        BoldPersistenceHandleFileXML1  
Property:         FileName  
Value:            'C:\Test.XML'

To begin with the XML file is hard coded, this of course could be set using code at runtime in a real application.

Component:        BoldSystemHandle1  
Property:         AutoActivate  
Value:            True

This property results in the Bold components opening the XML file and being ready for immediate use on application startup.

## Building the Model

The next step is to build our model. Bold for Delphi/Bold for C++ comes with an inbuilt non-graphical UML modeling tool. This is accessed by double clicking on the *BoldModel* component.

### Bold UML Model Editor

The bold UML model editor (Figure 1) contains the application model information, data type information, and relational database mapping information.

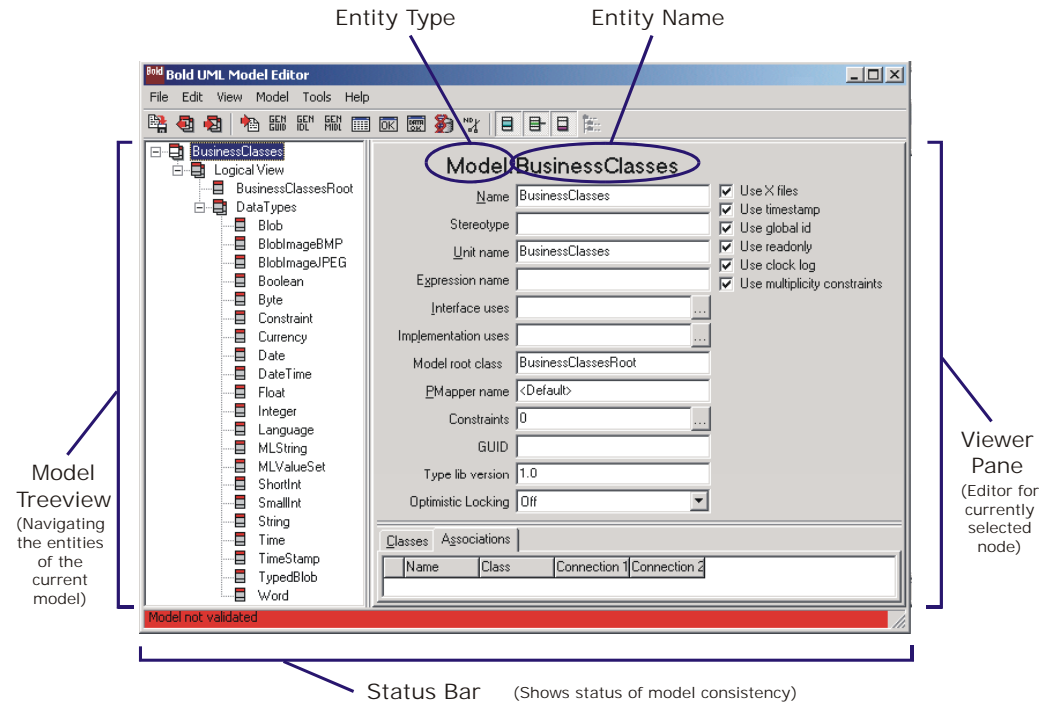


Figure 1: Model Editor

### Model Entities

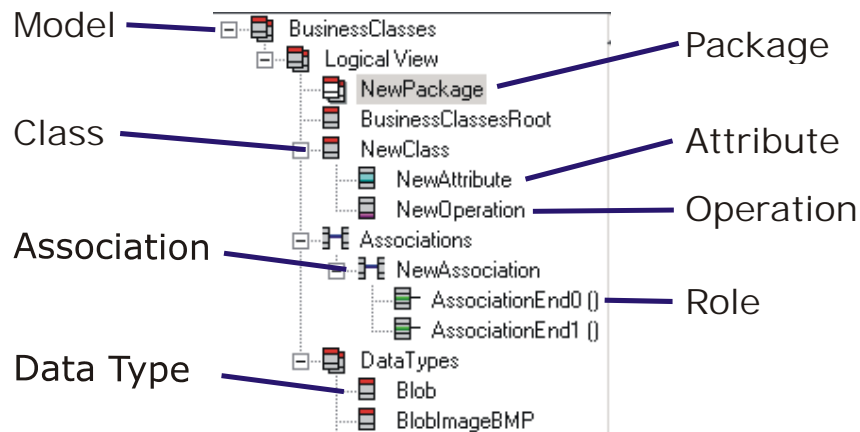
The modeler supports the following entity types:

- **Model:** Collection of all entities representing the problem domain.
- **Package:** A collection of entities representing a subset of the entire model, useful for reducing complexity by breaking a model into a collection of smaller models.
- **Class:** An object, similar in concept to Delphi classes (Delphi classes can be generated for an UML class) but can contains information and relationships not directly available to object pascal classes. Fortunately the Bold framework wraps this extra functionality into object pascal classes by using association classes and special list classes that remove this complexity for the pascal programmer.
- **Attribute:** This is the equivalent to Delphi properties, however in the Bold products these can be derived using an OCL expression or have initialization values set directly in the model. In a lot of circumstances

this removes the need to create getter and setter methods like in traditional object pascal programming.

- **Operation:** This is the equivalent to a method of a Delphi Class (Function/Procedure).
- **Association:** Associations represent the relationships between classes. Depending on the type of relationship an association maybe represented by a class itself, this allows for Operations & Attributes to be part of an association between classes. Associations in the Bold framework are much more powerful than properties of a class type in Delphi objects, again the complexity of mapping this extra functionality in Delphi classes is performed by the Bold framework.
- **Role:** A Role represents the connection of an Association to a class.
- **Data Type:** Represents the different Data Types supported within the model. This can be extended to include custom Data Types.

Figure 2 shows the how different entities are represented in the Model Editor. These will be explained in more detail as we build our model.



**Figure 2: Model Entity Types**

The *Model* menu in the editor will change depending on the entity selected, these options are also available from the right-click context menu in the Model Treeview. For example, to add a Class you need to have a Model entity or Package entity selected, as these are the only entities that can contain a Class.

Most of the options are fine 'out of the box' and this is how we will start with Bold for Delphi/Bold for C++.

In Figure 1 The root node *BusinessClasses* is selected and the viewer pane on the right shows the global options for our model, to begin with the most important are:

- **Name:** The name for the model.

- **Unit name:** The pascal unit that will be generated to hold our classes if we choose to generate code. Generating code is optional. However if you do generate code this is the default unit that the code is placed. This can be overridden for each class if desired.

For both these options we will leave the default values.

### Adding our Business Objects

With *BusinessClasses* highlighted, select the *New Class* option from the *Model* menu. This option is only available if a *model* entity is selected in the treeview.

Change the name of the newly created class to *Contact* and then repeat creating a new class for both *Company* and *Person*. Don't forget you will need to select the *BusinessClasses* model each time for the *New Class* menu item to be visible. (Optionally you can use the right-click context menu in the treeview)

Set the following options for the new classes:

- **Contact:** Set the abstract checkbox as checked.
- **Person:** Set the *SuperClass* combo box to *Contact*.
- **Company:** Set the *SuperClass* combo box to *Contact*.

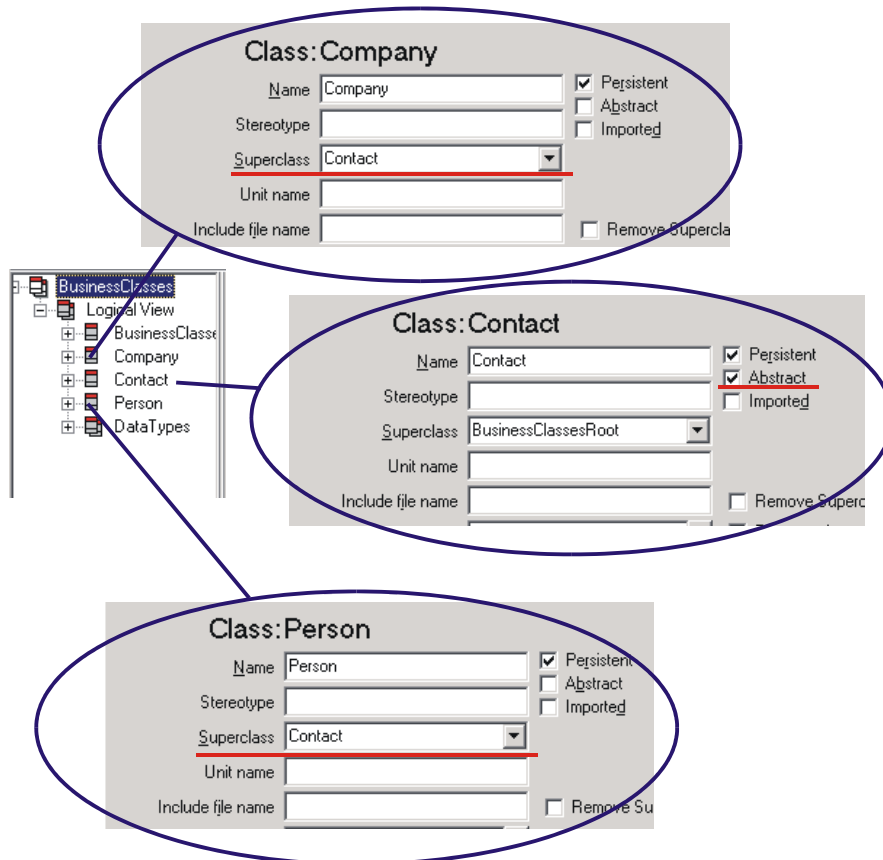


Figure 3: Basic Classes

Figure 3 shows the changes applied in the model editor. We have now created our basic classes, but more importantly we have enforced some of the business rules mentioned earlier.

By marking the *Contact* class as abstract, the Bold framework will not allow you to create instances of this class. Because both *Company* and *Person* have the superclass of *Contact*, they will inherit any attributes and operations of *Contact*. Importantly any instance of *Company* or *Person* can be treated as if they are a class of *Contact*, making it possible, where required to treat these classes the same.

#### *Adding Attributes*

Select the *Contact* class in the model editor treeview. Using the *Model* menu again you'll notice the options have changed to suit a class entity. Select *Add Attribute* three times and name them *name*, *address* and *phoneNumber* respectively. You will notice the capitalization I have used is a bit different. This is an UML convention, although not enforced in the model, it is a good habit to follow. When writing OCL expressions the UML style capitalization's are important and errors will result form poorly formed expressions.

Check each attribute as *Persistent*, this is required for Bold for Delphi/Bold for C++ to save the values to the database. Leave the default type as *String* and leave the default length of 255. Strings are the only type that recognizes the length parameter.

Because we have added these attributes to the *Contact* class and both *Company* and *Person* descend from *Contact*, they will each inherit the attributes of *name*, *address* and *phoneNumber*.

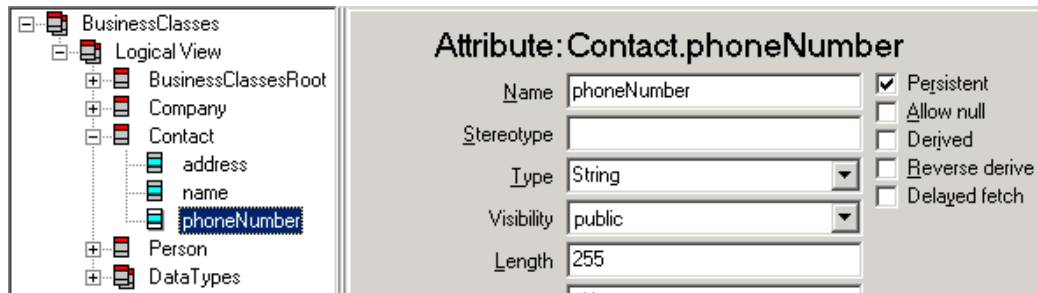


Figure 4: Basic Attributes

#### *Adding the Association*

With *BusinessClasses* highlighted, select the *New Association* option from the *Model* menu. This option is only available if a *model* entity is selected in the treeview.

Name the new association *Employment* and name each of the association ends (Roles) *worksFor* and *employs*. For each of the roles set the following parameters:

- **worksFor**: Select *Company* as the class for this role.

- **employs:** Select *Person* as the class for this role. Unselect the *Embed* checkbox and choose *0..\** as the multiplicity for this role. You will see on changing the multiplicity that the Bold framework automatically checks the *Multi* checkbox.

These changes are shown in Figure 5.

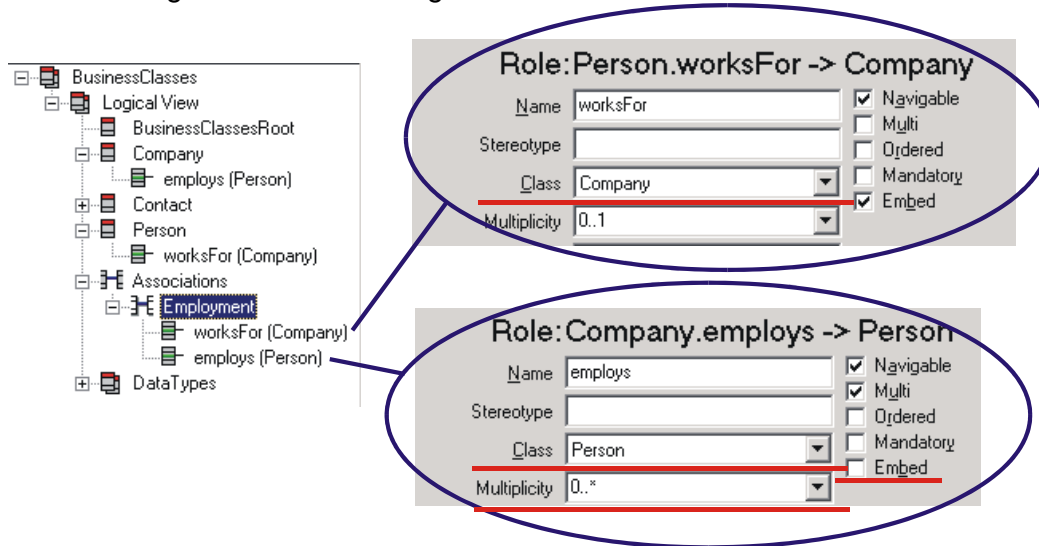
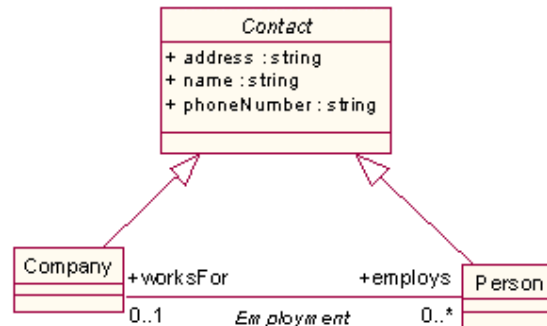


Figure 5: Employment Association

This is all that is required to complete the modeling of our original design. Using the Bold framework we have accurately represented all shown business rules simply and without complicating the design due to language limitations.



Before we leave the model editor lets have a look at *multiplicity* and *embed* options a little further.

Multiplicity is the measure of the number of 'connections' at the association end point. There are four basic options:

- **0..1:** Zero or one connection.
- **1..1:** Must have one and only one connection.
- **0..\*:** Zero or many connections.
- **1..\*:** One or many connections.

It is also possible to use n-cardinality relationships like 4..7 or 2..2. Embedding is the process of using a field in association end's class to store the associated objects reference, for now all you need to know is that:

- To be able to embed an association end it must single cardinality. E.g. 0..1 or 1..1.
- If both associations can have more than two connections then you must use a class to hold the association. This is also true if you decide not to embed a both ends of a single cardinality association.
- It is not good practice to embed both ends of an association.

This topic will be revisited in more detail later.

## Adding a User Interface

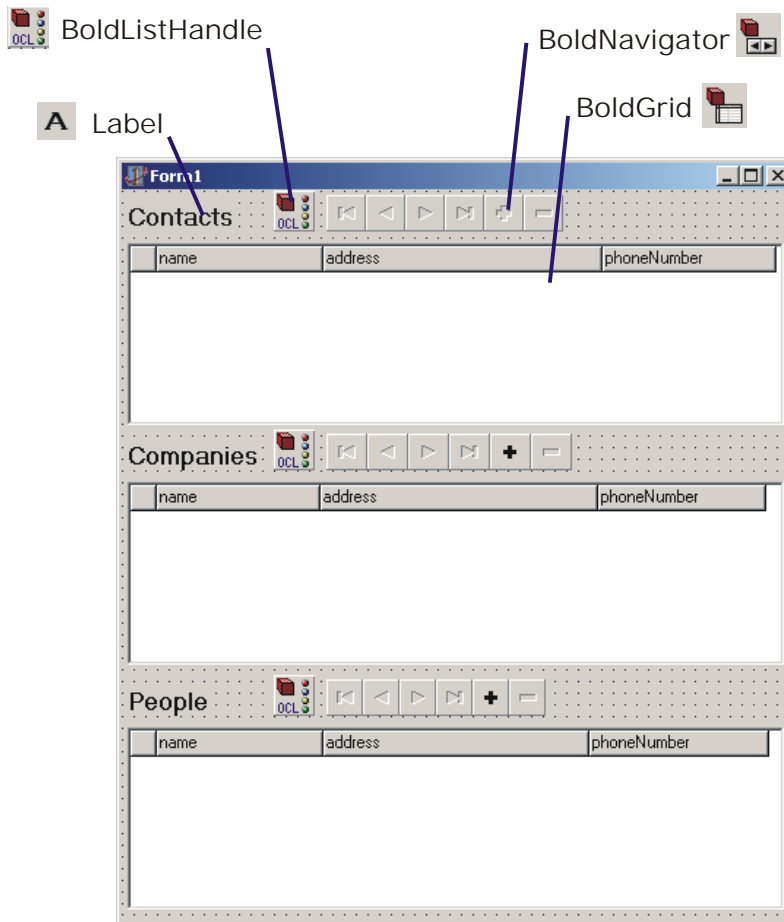
### Simple Main Form

To get the application up and running as soon as possible we will build a simple main form allowing us to edit the business objects. This will allow for testing of the design and a demonstration of a few Bold for Delphi/Bold for C++ features.

To the **MainForm.pas** unit add **BoldAFPDefault** and **BusinessLayer** to the uses clause. The **BoldAFPDefault** unit allows Bold for Delphi/Bold for C++ to automatically generate editor forms for the business objects that can be launched automatically from Bold-aware controls. The **BusinessLayer** unit allows the Bold Controls to see the model on the data module.

From the **Bold Handles** component palette add 3 **BoldListHandle** components to the form. From the **Bold Controls** component palette add 3 **BoldNavigator** components and 3 **BoldGrid** components. Also add 3 standard Delphi **Label** components to help keep everything in order.

Use the following layout to position the controls. The caption of each label has been set to describe the control:



For each group of controls set the following properties:

### **Contacts**

#### *BoldListHandle*

Name: ContactList  
RootHandle: BusinessModule.BoldSystemHandle1  
Expression: Contact.allInstances  
Enabled: True

#### *BoldNavigator*

Name: ContactNavigator  
BoldHandle: ContactList  
DeleteQuestion: 'Delete Contact?'  
Enabled: True

#### *BoldGrid*

Name: ContactGrid  
BoldHandle: ContactList

After setting the **BoldHandle** property, right-click on the grid and select **Create Default Columns** from the context menu.

### **Companies**

#### *BoldListHandle*

Name: CompanyList  
RootHandle: BusinessModule.BoldSystemHandle1  
Expression: Company.allInstances  
Enabled: True

#### *BoldNavigator*

Name: CompanyNavigator  
BoldHandle: CompanyList  
DeleteQuestion: 'Delete Company?'  
Enabled: True

#### *BoldGrid*

Name: CompanyGrid  
BoldHandle: CompanyList

After setting the **BoldHandle** property, right-click on the grid and select **Create Default Columns** from the context menu.

### **People**

#### *BoldListHandle*

Name: PersonList  
RootHandle: BusinessModule.BoldSystemHandle1  
Expression: Person.allInstances  
Enabled: True

#### *BoldNavigator*

Name: PersonNavigator  
BoldHandle: PersonList

DeleteQuestion: 'Delete Person?'  
Enabled: True

### *BoldGrid*

Name: PersonGrid  
BoldHandle: PersonList

After setting the **BoldHandle** property, right-click on the grid and select **Create Default Columns** from the context menu.

Almost ready to run, add the following event handler for the form's **OnClose** event:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    BusinessModule.BoldSystemHandle1.UpdateDatabase;  
end;
```

This ensures that any information entered at runtime will be saved to the XML file.

## Running The Application

### Entering Test Data

Run the application and enter some companies and people. Figure 6 shows the application at runtime with some sample data.

'New Contact' disabled

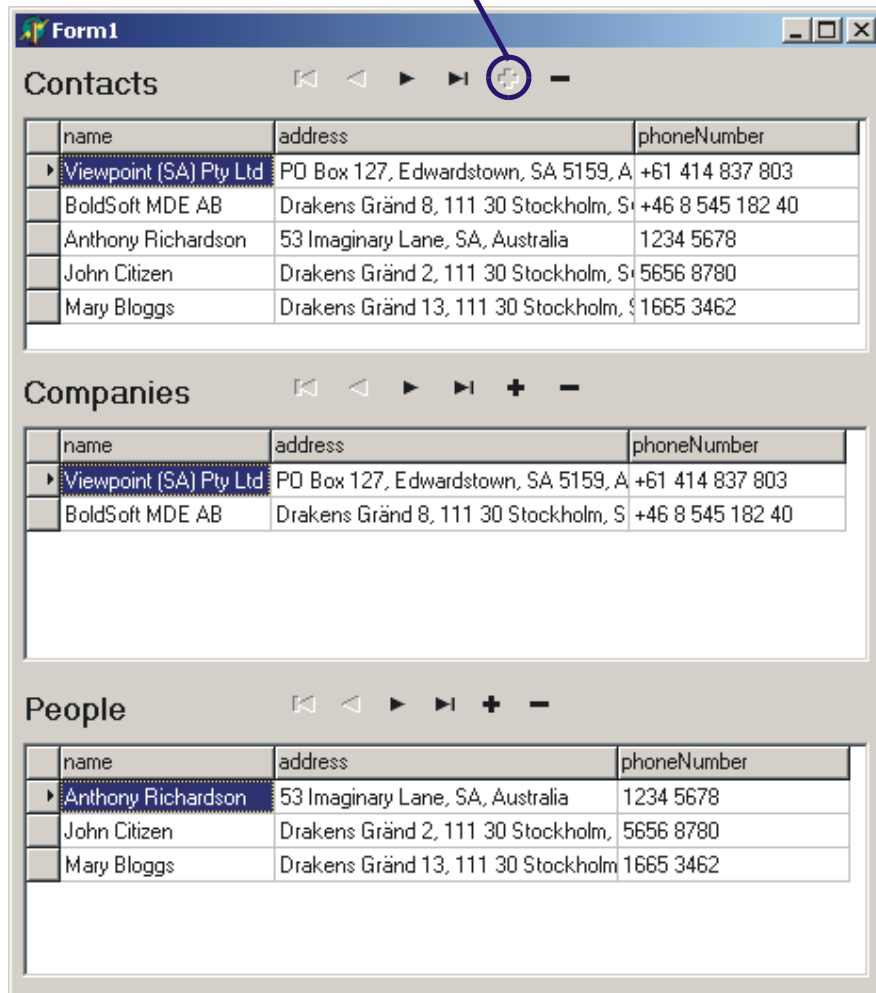


Figure 6: Basic Application at Runtime

It is important to spend sometime discussing what the features of this basic screen are in comparison to traditional Delphi database applications. The distinctions are important because the Bold for Delphi/Bold for C++ data-aware solution overcomes many shortcomings of traditional Delphi data-aware controls.

## ***Delphi vs. Bold Data Aware Comparison***

There are some important behavioral differences between Delphi data-aware and Bold-aware components. The key to the difference is meta-data, that is data about data. This may be confusing at first but it's the meta-data in the model driven Bold for Delphi/Bold for C++ approach that allows for the more sophisticated controls in a Bold-aware application.

A simple example is the **Contacts Navigator**, it recognizes that **Contacts** are an abstract class and cannot be created directly. Therefore it disables the '+' append button (As indicated in Figure 6). Although only a simple example, the benefits of controls that understand the context of the data they are presenting is extremely valuable and assists in writing correct code.

Another important distinction between Bold-aware and standard data-aware solutions is the synchronization of data between grids. In a traditional data-aware application you would need to resort to using:

### *A) Multiple Queries*

Three grids each linked to a query containing the SQL statement required to retrieve the desired data.

This solution is relatively easy to implement but suffers the problem of no synchronization between each grid. Updating each grid requires re-running the queries. This can be problematic and requires careful consideration with the status of each query (e.g. Is it currently in edit mode?).

### *B) Filtered Grids*

By using a single query and purchasing a third-party grid with inbuilt filtering it may be possible to replicate the functionality without resorting to desynchronizing queries. However depending on the nature of the SQL statement it may not be updateable, leaving you with a read-only solution. Another, not so obvious, limitation of this method is that a single record only can be the 'active' or 'current' record. Depending on the third-party grid supplied this can cause problems.

*“There are some important behavioral differences between Delphi Data-aware and Bold-aware components”*

In Figure 6 that each grid has a separate object selected and entering or changing data the changes are automatically propagated to all other controls. Bold for Delphi/Bold for C++ supports various methods to ensure this behavior is supported between applications running on different computers across the network. A change in an object on one computer can automatically be reflected and updated on every other computer on the network.

More examples of the differences will eventuate as the series progresses.

## Bold Object Editor Forms

With the application running double click the index marker on the left hand side of a grid. An object editor dialog will be created, customized to the object type. These editors are enabled by the inclusion of the **BoldAFPDefault** unit anywhere in the application.

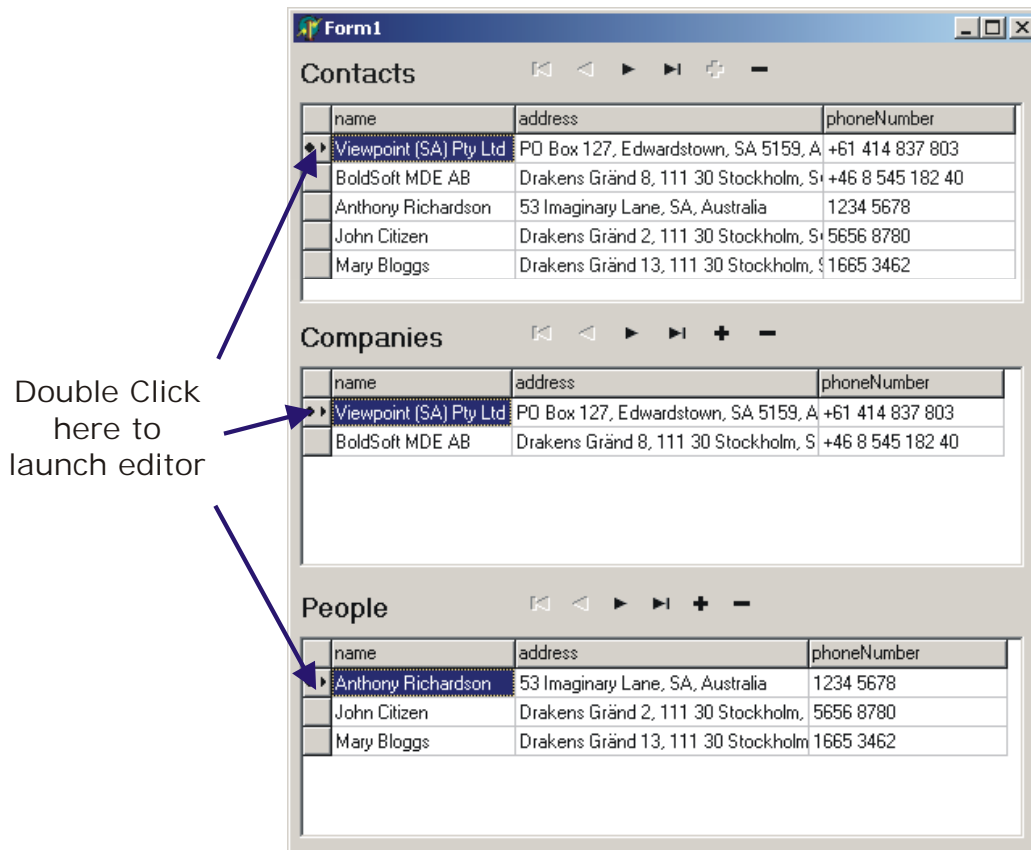


Figure 7: Editor Launch Points

In Figure 7 the selection column of the grid displays a small circle to indicate that the object in that row is selected and a small arrowhead to indicate that the object for that row is the currently active object. It is possible to multiple select objects (rows) in the grid by *ctrl+clicking* each object. It is also possible to select all objects in the grid by clicking the top left cell in the column header.

The default object editors aren't much to look at but they are fully functional and support all relationships. You may have noticed that our application was incapable of displaying and editing the relationships between companies and people. The inbuilt editors support these relationships.

Using the Drag Points on the editors (refer Figure 8) you can establish the links by dragging a **Person** into the employs grid of the **Company** object.

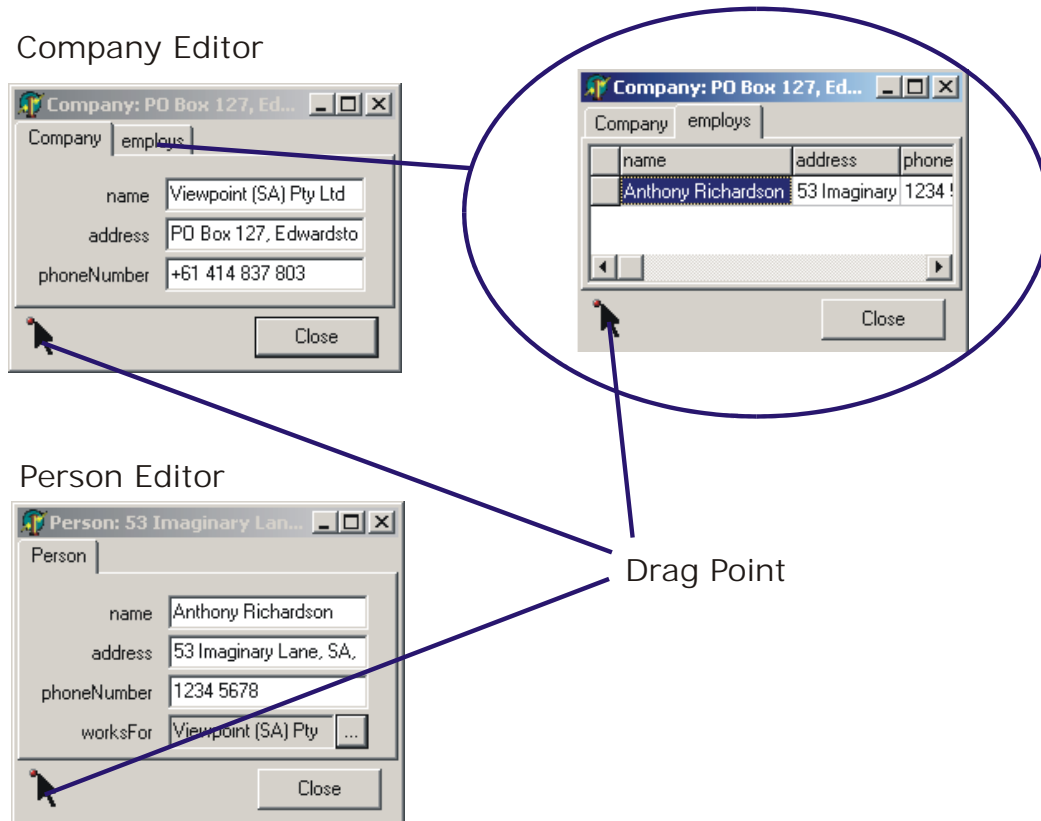


Figure 8: Object Runtime Editors

It is also important to note the other differences with these editors. They are not linked to the grid so you can have multiple editors open and navigate the grid without problem. It is problematic in a traditional data-aware application to do this as when the cursor changes the edit controls change.

## **Summary**

This article has demonstrated the basics a building a Bold Model driven application. It has shown the key concepts of a simple Bold Application and discussed briefly the usage of these concepts. Briefly a comparison to traditional database programming was supplied where possible.

As much as this article has introduced the basics, many questions remain and only a small part of the Bold for Delphi/Bold for C++ product has been shown.

## ***Future***

Future articles will explore some of the existing concepts in more depth and venture into the specifics of using various Bold constructs to solve the common challenges faced in application development.