

Starting with Bold for Delphi/Bold for C++ Part 2: Extending Models

*By Anthony Richardson
31 October 2001
Revision 1.0*

Contents

OVERVIEW.....	3
Introduction.....	3
About the Author	3
Acknowledgments	3
Credits	3
Part 2: Extending Models	3
GETTING STARTED	4
Example Application.....	4
Build Basic Project	5
Build the Model.....	6
Generating Pascal Code	10
Adding a User Interface	11
Running The Application.....	15
USING CODE IN BOLD APPLICATIONS	18
Enhancing the application with code.....	18
INTRODUCTION TO OBJECT LIFECYCLES.....	20
Using Delphi Code	20
Model driven object life management	21
DERIVED & REVERSE DERIVED ATTRIBUTES.....	23
Adding the new attributes.....	23
Reversed Derived.....	25
Derived Attributes with OCL.....	27
OPERATIONS	29
Adding an Operation	29
DERIVED RELATIONSHIPS.....	33
Adding the Managed By relationship	34
Adding the Workforce relationship	35
ONE-WAY RELATIONSHIPS	36
CONSTRAINTS: A PARTING NOTE.....	42
SUMMARY	42

Overview

Introduction

'Starting with Bold for Delphi/Bold for C++' is a series of articles design to provide an introduction to the Bold for Delphi/Bold for C++ product 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++ product 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. More information about Viewpoint (SA) Pty Ltd is available on the Internet at 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 2: Extending Models

In the first article of this series the Bold model editor was introduced and basic modeling concepts explained. However, there is a lot that can be achieved within the Bold Model editor and the purpose of this article is to explorer the concepts of:

- Operations: Add business logic to our classes.
- Derived Attributes: Calculating attribute values 'on the fly'.
- Reverse Derived Attributes: Using code to both compose and decompose derived values within your class.
- Derived Relationships: Resolving relationships 'on the fly'.

Getting Started

Example Application

To begin the journey into more advanced modeling techniques we will start with a model for a basic Human Resource Management Application.

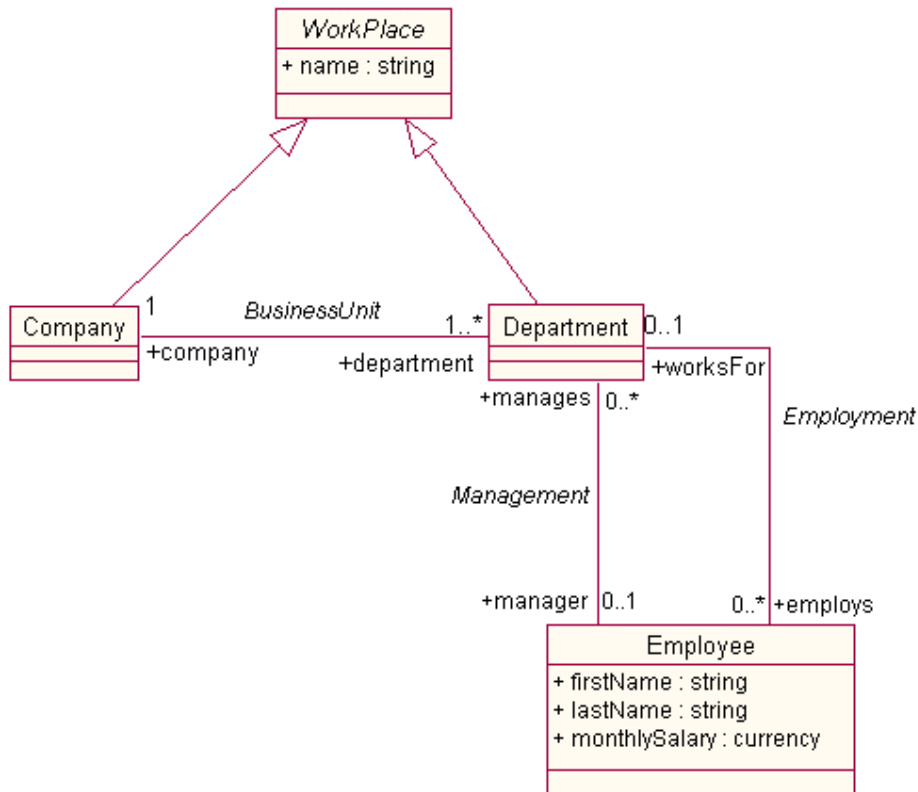


Figure 1: Basic Class Model

The first article in this series covered the construction of a basic model. The concept of classes, attributes and relationships was discussed. In Figure 1 no new concepts are introduced and the following basic business rules are enforced:

- We have Company and Departments, which are types of Workplace.
- We have Employees, which can work for a department
- Companies can contain Departments.
- A Department can have one manager, which is an Employee.
- An Employee can manage multiple departments

We will step through the creation of this with a basic GUI quickly so we can get to the real interesting modeling concepts sooner.

Build Basic Project

Groundwork

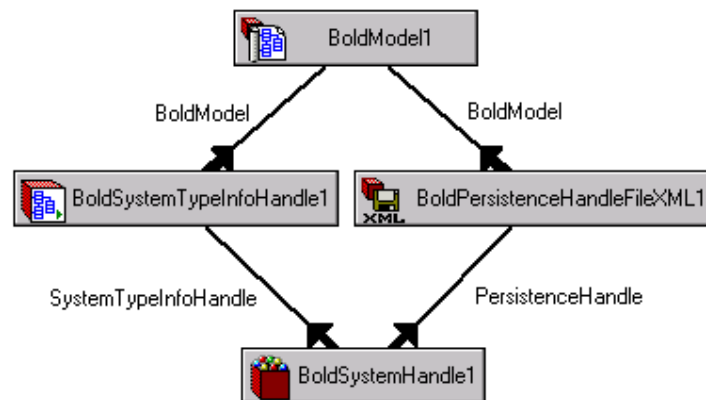
To get started we need to setup a basic Delphi project with a data module.

1. Select *File|New Application* from the Delphi menu.
2. Save the Form unit as *MainForm.pas* and the project as *HRManager.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

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*.
4. From the *Bold Persistence* component tab add a *BoldPersistenceHandleFileXML* component to the data module.
5. Link the *BoldModel* property to the *BoldModel* component.
6. Link the *PersistenceHandle* property of *BoldSystemHandle* to the *BoldPersistenceHandleFileXML* component.



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: True

This allows us to work directly with the model using Object Pascal.

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

Component: BoldSystemHandle1
Property: AutoActive
Value: True

This property results in the BoldSystemHandle1 opening the XML file and being ready for immediate use on application startup.

Add the following event handler for the data module's **OnDestroy** event:

```
procedure TBusinessModule.DataModuleDestroy(Sender: TObject);  
begin  
    BoldSystemHandle1.System.UpdateDatabase;  
end;
```

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

Build the Model

Double click the BoldModel component to open the Bold Model Editor. Set the model *name* to HRClasses as per Figure 2. Change models *Unit name* to HRClasses and *Model root class* value to HRClassesRoot. Change the BusinessClassesRoot classes *name* to HRClassesRoot.

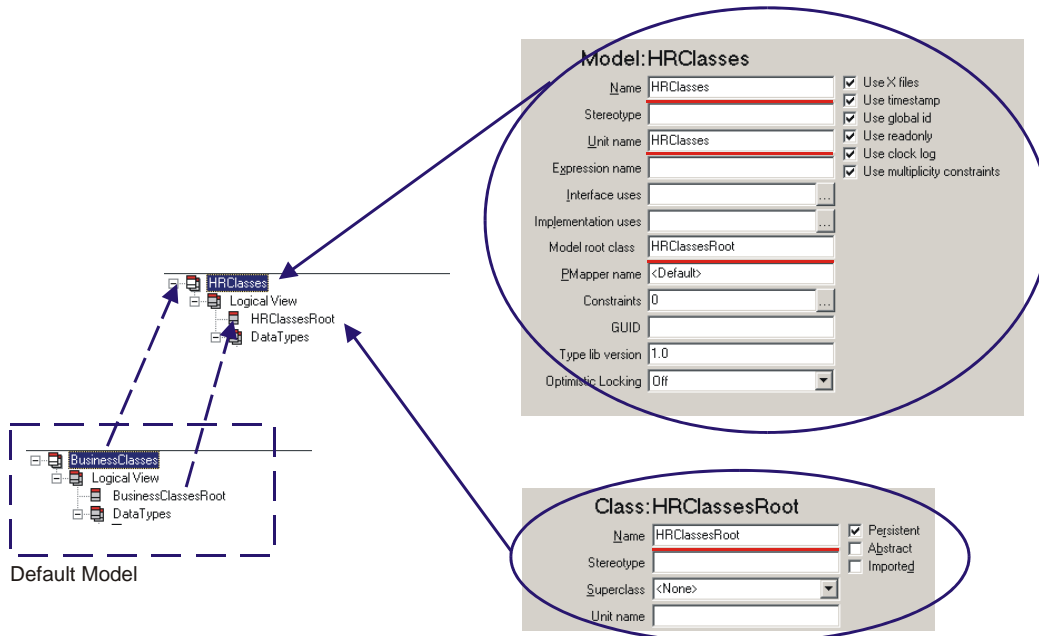


Figure 2: Model Properties

Changing the models default values above should become a habit. Leaving the default names may cause problems in future projects. When generating

source code Bold for Delphi/Bold for C++ will search the current project directory, directories of other files in the project and the Delphi search path for files matching *Unit name*. If the file is found it will be overwritten. This is the desired behavior if the file is for the current project but can cause problems if working with multiple models in related projects.

For the reason given above it is also a good idea to ensure the project is saved to a directory before generating code from your model.

Construct the classes and set the properties as indicated:

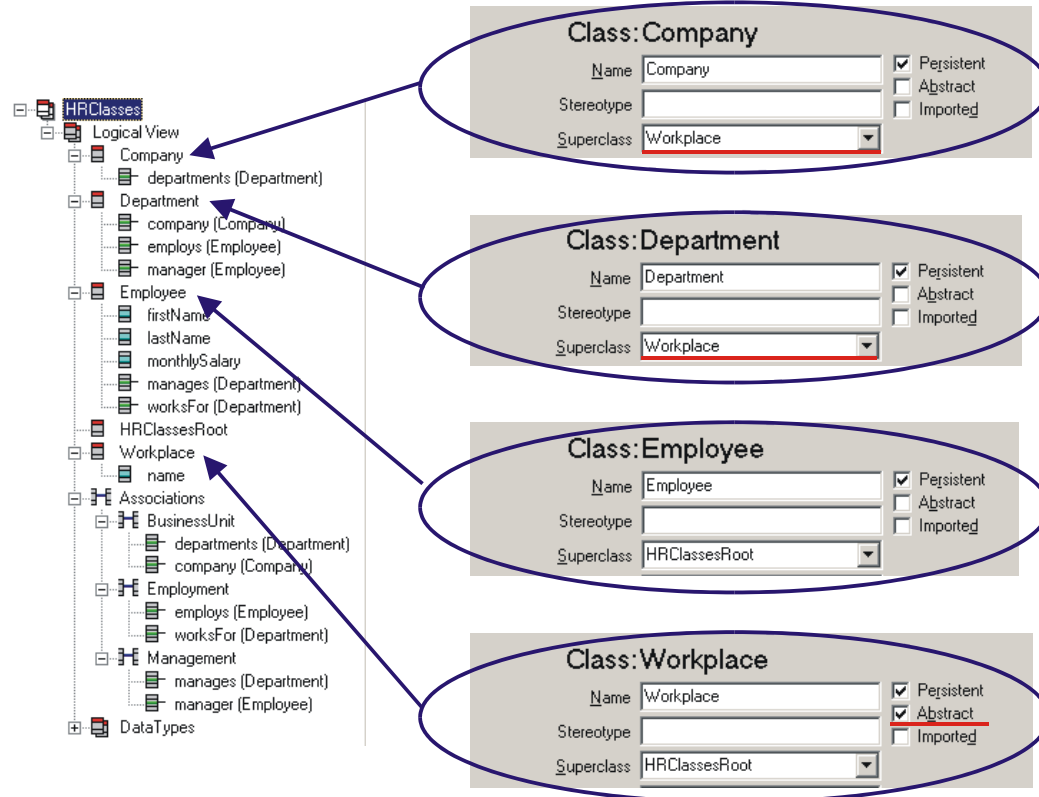


Figure 3: Basic Classes

And the attributes:

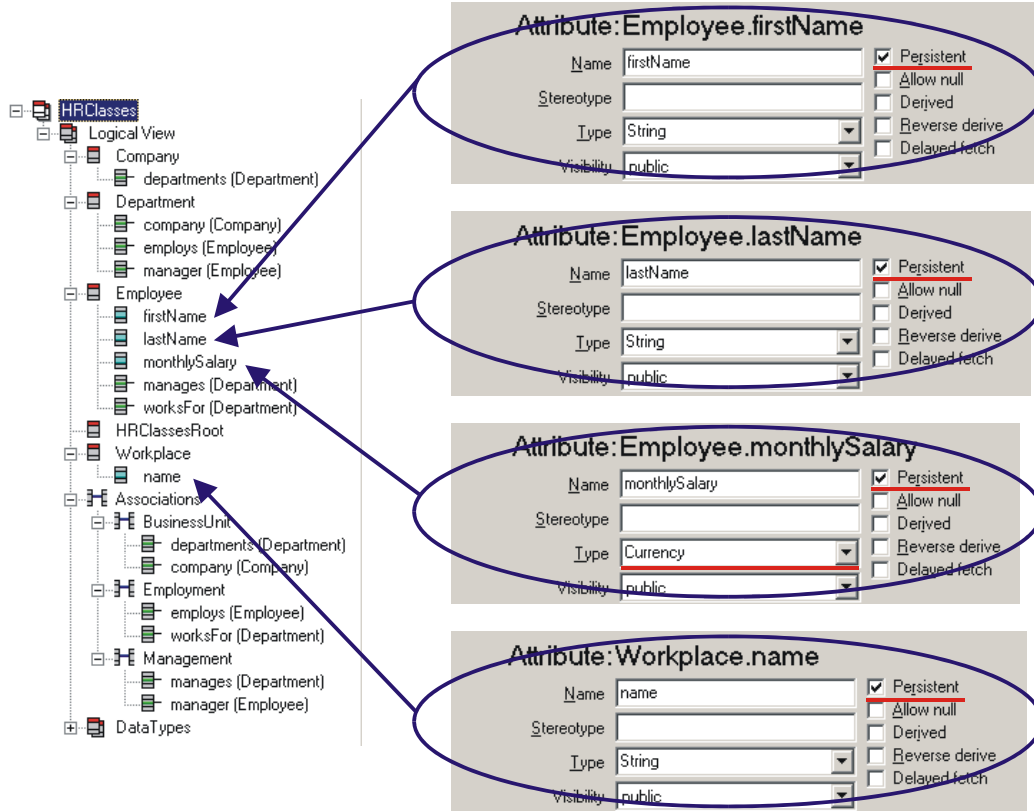


Figure 4: Basic Attributes

And the associations:

The image displays a screenshot of the Delphi IDE's class browser on the left and several configuration panels for associations on the right. The class browser shows a hierarchy starting with 'HRClasses' and 'Logical View', containing classes like 'Company', 'Department', 'Employee', and 'Workplace'. Under 'Associations', it lists 'BusinessUnit', 'Employment', and 'Management'. The configuration panels on the right are arranged vertically and connected to the class browser by blue arrows. Each panel shows the configuration for a specific association, including its name, class, multiplicity, and various flags like 'Persistent', 'Navigable', and 'Embed'.

Association: BusinessUnit

- Name: BusinessUnit
- Class: <none>
- Constraints: 0
- Flags: Persistent, Derived

Role: Company.departments -> Department

- Name: departments
- Class: Department
- Multiplicity: 0..*
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Role: Department.company -> Company

- Name: company
- Class: Company
- Multiplicity: 1
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Association: Employment

- Name: Employment
- Class: <none>
- Constraints: 0
- Flags: Persistent, Derived

Role: Department.employs -> Employee

- Name: employs
- Class: Employee
- Multiplicity: 0..*
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Role: Employee.worksFor -> Department

- Name: worksFor
- Class: Department
- Multiplicity: 0..1
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Association: Management

- Name: Management
- Class: <none>
- Constraints: 0
- Flags: Persistent, Derived

Role: Employee.manages -> Department

- Name: manages
- Class: Department
- Multiplicity: 0..*
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Role: Department.manager -> Employee

- Name: manager
- Class: Employee
- Multiplicity: 0..1
- Flags: Navigable, Multi, Ordered, Mandatory, Embed

Figure 5: Basic Associations

Generating Pascal Code

In the Bold Model Editor select **Generate Code** from the **Tools** menu. You will be prompted to confirm or change the default include file created to hold class definitions. Accept the default, HRClasses_Interface.inc name. You will then be prompted with a name for the unit to store class information, again accept the default.

The generated code files will be loaded into the Delphi code editor as in Figure 6. The use of include files in the Bold framework is to ensure that code completion works correctly. The generated code files (*.pas & *_Interface.inc) should not be altered, as they will be overwritten on next code generation. Other generated code files (*.inc, excluding *_Interface.inc) will only be appended to; these are generated when using operations or derived attributes/relationships.



```
HRClasses.pas
MainForm | BusinessLayer | HRClasses | HRClasses_Interface.inc
(*****)
(*      This file is autogenerated      *)
(*  Any manual changes will be LOST!  *)
(*****)
(* Generated 26/10/2001 10:51:41 PM    *)
(*****)
(* This file should be stored in the   *)
(* same directory as the form/datamodule *)
(* with the corresponding model       *)
(*****)
(* Copyright notice:                  *)
(*                                    *)
(*****)

unit HRClasses;

{$DEFINE HRClasses_unitheader}
{$INCLUDE HRClasses_Interface.inc}

uses
  {ImplementationUses}
  {ImplementationDependancies}
  BoldGeneratedCodeDictionary;

{ Includefile for methodimplementations }
```

Figure 6: Generated Code

It is interesting to have a look at the generated classes.

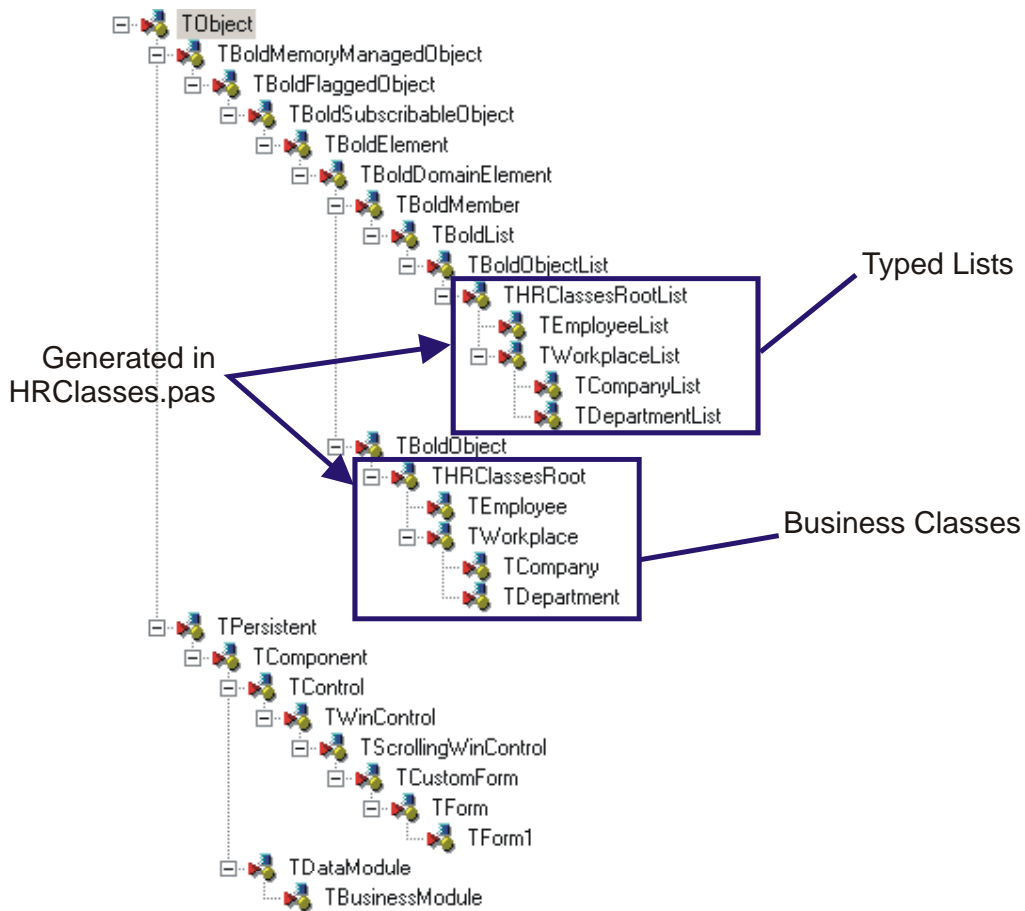


Figure 7: Class Hierarchy

Figure 7 shows the class hierarchy for the project. The classes marked in a blue outline are the auto-generated classes in HRClasses.pas. You will notice for each class created that the Bold framework automatically creates a matching typed list. This allows for clean code when working with lists of objects and multi-relations, the full benefit of compiler type checking is available to your code. Bold for Delphi/Bold for C++ also maintains the model hierarchy allowing for normal Delphi/C++ class comparisons when working with objects in code.

Adding a User Interface

Simple Main Form

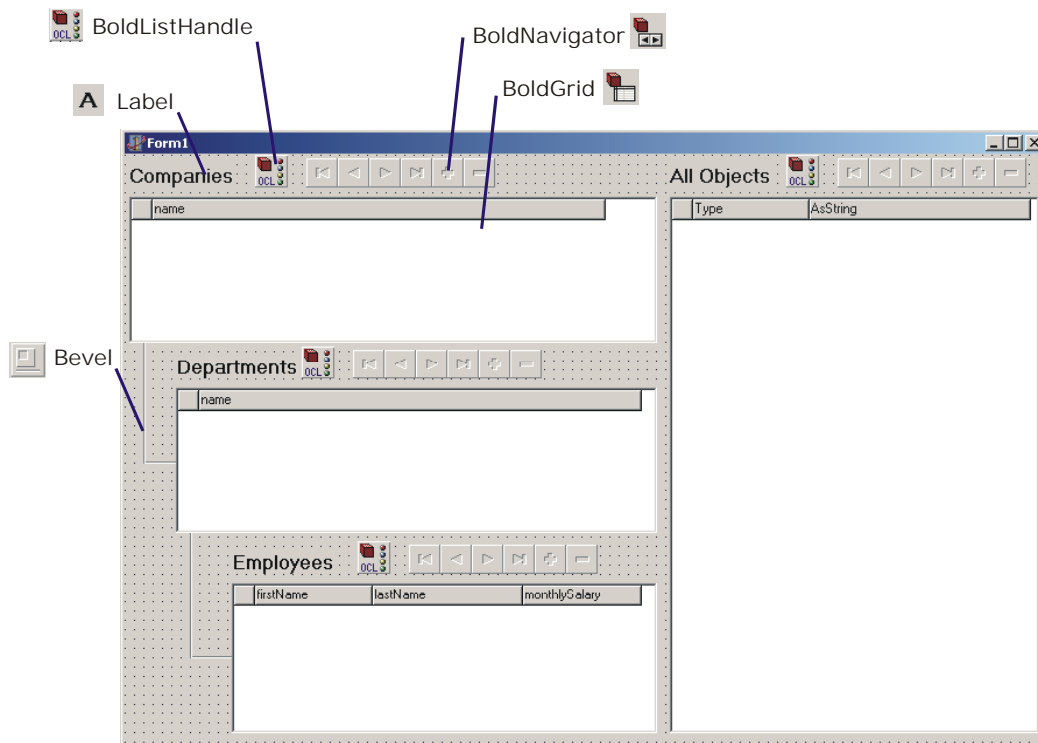
To get the application up and running as soon as possible we will build a simple main form allowing us to edit the business objects. This will allow for testing of the design and a demonstration of a few Bold for Delphi/Bold for C++ features.

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 4 **BoldListHandle** components to the form. From the **Bold Controls** component palette add 4 **BoldNavigator** components and 4 **BoldGrid** components. Also add 4 standard Delphi **Label** & **Bevel** 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:

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.

Departments

BoldListHandle

Name: DepartmentList

RootHandle: CompanyList

Expression: departments

Enabled: True

Using the **CompanyList** handle as the root handle allows the **DepartmentList** handle to create a master detail relationship. The expression **department** means that this list will automatically contain all departments linked to the currently selected company.

BoldNavigator

Name: DepartmentNavigator

BoldHandle: DepartmentList

DeleteQuestion: 'Delete Department?'

Enabled: True

BoldGrid

Name: DepartmentGrid

BoldHandle: DepartmentList

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

Employees

BoldListHandle

Name: EmployeeList

RootHandle: DepartmentList

Expression: employs

Enabled: True

Using the **DepartmentList** handle as the root handle allows the **EmployeeList** handle to create a master detail relationship. The expression **employs** means that this list will automatically contain all employees employed by the currently selected department.

BoldNavigator

Name: EmployeeNavigator

BoldHandle: EmployeeList

DeleteQuestion: 'Delete Employee?'

Enabled: True

BoldGrid

Name: EmployeeGrid

BoldHandle: EmployeeList

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

All Objects

BoldListHandle

Name: ObjectList
RootHandle: BusinessModule.BoldSystemHandle1
Expression: HRClassesRoot.allInstances
Enabled: True

BoldNavigator

Name: ObjectNavigator
BoldHandle: ObjectList
DeleteQuestion: 'Delete Object?'
Enabled: True

BoldGrid

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

Running The Application

When you run the application add some entries for Company, Department and Employees. The main form will look similar to Figure 8.

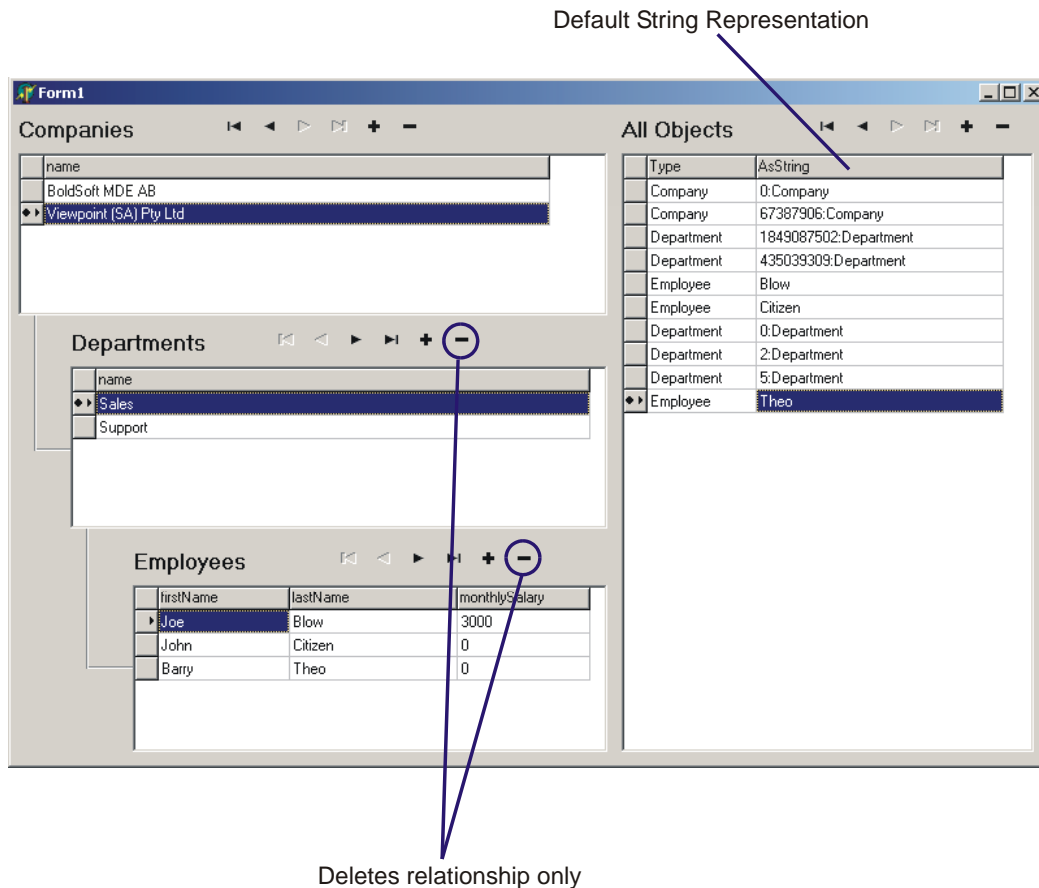


Figure 8: Basic Application at Runtime

When you add a new department, using the BoldNavigator, Bold for Delphi/Bold for C++ will automatically establish the relationship with the company, likewise adding employees adds the relationship to departments. When you delete from the Company grid or the All Objects grid the object is deleted. When you delete from the Department or Employee grid the object is **not** deleted, but the relationship is unlinked. This change in behavior is the default when you cascade list handles. This can be changed by altering the *BoldDeleteMode* property of the *TboldNavigator*. It is possible to set this to *dmUnlinkAllandDelete* to force the actual deletion of the object.

Double clicking a row will open the default editor for that object and allow drag and drop relationship linking.

The all objects grid is interesting; here the Bold framework has created a column to display the type of the object and a column to display the object 'AsString'. The AsString column is the string representation of the object. In the example the Bold framework has automatically used the *lastName* of employees to represent them but has created a mangled string for both

companies and departments. The reason for this is both the company and department classes don't have any attributes (they rely on inheritance). Since this is confusing we can tell Bold for Delphi/Bold for C++ how to represent the objects in a more human readable way.

Default String Representation

Figure 9 shows the settings for each class. By using an OCL expression a human readable identifier can be created for each class.

For the Company class we simply use the company *name* as inherited from the Workplace class. For Department we grab the *name* of the *company* associated with the department, add a hyphen and append the department *name*. For the Employee class the aggregation of the *firstName* and *lastName* are used.

The figure consists of three vertically stacked screenshots of class configuration dialogs. Each dialog has a title bar and several input fields and checkboxes.

- Class: Company**
 - Name: Company
 - Stereotype: (empty)
 - Superclass: Workplace
 - Unit name: (empty)
 - Include file name: (empty)
 - Default string rep: name
 - Table mapping: Own
 - Checkboxes: Persistent (checked), Abstract (unchecked), Imported (unchecked), Remove Superclass on unboldify (unchecked), Remove class on unboldify (unchecked), Is Root Class (unchecked)
- Class: Department**
 - Name: Department
 - Stereotype: (empty)
 - Superclass: Workplace
 - Unit name: (empty)
 - Include file name: (empty)
 - Default string rep: company.name + '-' + name
 - Table mapping: Own
 - Checkboxes: Persistent (checked), Abstract (unchecked), Imported (unchecked), Remove Superclass on unboldify (unchecked), Remove class on unboldify (unchecked), Is Root Class (unchecked)
- Class: Employee**
 - Name: Employee
 - Stereotype: (empty)
 - Superclass: HRClassesRoot
 - Unit name: (empty)
 - Include file name: (empty)
 - Default string rep: firstName + ' ' + lastName
 - Table mapping: Own
 - Checkboxes: Persistent (checked), Abstract (unchecked), Imported (unchecked), Remove Superclass on unboldify (unchecked), Remove class on unboldify (unchecked), Is Root Class (unchecked)

Figure 9: Setting Default String Representation

This results in a much more readable display as shown in Figure 10.

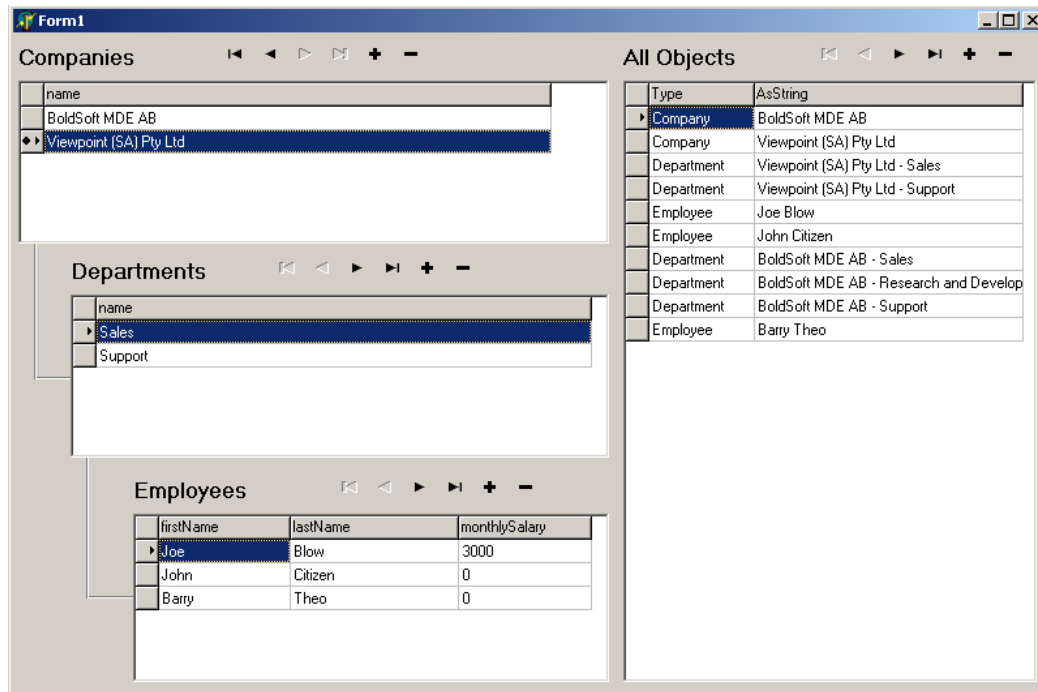


Figure 10: Application at Runtime (with readable object string representation)

Before we move into write Pascal code let's add a new column to the department grid and display the manager for the department.

1. Right-click the DepartmentGrid and select 'Edit Columns'.
2. Add a new column in the column editor
3. Set the *BoldProperties.Expression* parameter to 'manager' (without the quotes).

Now when you assign a manager to the department the employee will be displayed in the grid. Because the result of the expression manager actually returns an employee object, the Bold Grid will actually use the employee classes *Default String Representation* set earlier. This results in the employee's full name being displayed in the grid. (Hint: Use the default forms and *drag and drop* to set the manager for a department)

Using Code in Bold Applications

Yet another powerful aspect of Bold for Delphi/Bold for C++ is the ability to use native Pascal objects for your business objects. When using traditional database programming the data has no type within the code you are writing (except for basic types like string or integer). When using the *Generate Code* functionality of Bold for Delphi/Bold for C++ you can use native Delphi classes that wrap the Bold Classes you have create in the Bold Modeler.

The benefits of being able to do this are enormous in terms of:

- Code readability.
- Code maintainability.
- Code accuracy.
- Development speed.

When coding with Bold for Delphi/Bold for C++ you inherently gain the benefit of Delphi's strong type-checking compiler. Delphi's integrated code editor prompts you with all your classes' information via code completion. You code reads closer to the level of intent of what you are trying to achieve, that means code is a lot closer to becoming self documenting.

Enhancing the application with code

We are going to add a pop-up menu to the employee grid and add the ability to delete, add new and promote an employee to manager.

- 1) Add **HRClasses** to the MainForm's unit uses clause. This is required to allow using the Bold classes in the main form.
- 2) Add a **TPopupMenu** component to the form and set it's name to **EmployeePopupMenu**.
- 3) Set the **EmployeeGrid**'s **PopupMenu** property to this new popup menu.
- 4) Add a menu item to the popup menu with the caption 'Make Manager'.

This sets up the basic framework for adding our code. Begin by setting the **EmployeeGrid**'s **OnContextPopup** event with the following code:

```
procedure TForm1.EmployeeGridContextPopup(Sender: TObject;  
    MousePos: TPoint; var Handled: Boolean);  
begin  
    // Enable/Disable Menu items depending if an Employee is  
    // selected.  
    MakeManager1.Enabled := Assigned(EmployeeGrid.CurrentBoldElement);  
end;
```

This code controls the availability of the popup menu item depending on if an object is available in the appropriate grid. If no object is selected in a grid the grid's *CurrentBoldElement* property will return *nil*. If an object is selected this property will return that item as a *TBoldElement*, this is a base class for all classes created from the model, and all lists of classes as well. (Refer Figure 7 from earlier in this article).

Add an *OnClick* event handler for the 'Make Manager' menu item.

```
procedure TForm1.MakeManager1Click(Sender: TObject);  
var CurrentEmployee: TEmployee;  
begin  
    // Check that the object is an Employee object and  
    // assigned it to our local variable  
    if EmployeeGrid.CurrentBoldElement is TEmployee then  
        begin  
            CurrentEmployee := TEmployee(EmployeeGrid.CurrentBoldElement);  
  
            // Check if the employee works for a department and set the  
            // employee as the manager  
            if assigned(CurrentEmployee.worksFor) then  
                CurrentEmployee.worksFor.manager := CurrentEmployee;  
        end;  
end;
```

The comments in the code should be almost unnecessary. The power of working with native Pascal code is evident. Even the mundane task of validating the selection is simplified using the Delphi **is** operator and a typecast to a local variable. The sheer simplicity of using a statement like 'CurrentEmployee.worksFor.manager := CurrentEmployee' in terms of maintainability and readability is incredible. This can be read as 'For the current employee, set the manager of the department for which they work to that employee'.

It would be good object orientated design to move this code into the *Employee* class as an operation call *Promote*. Then any error checking would be encapsulated in one location and any business rules about managing the promotion process is abstracted from the presentation layer.

Introduction to Object lifecycles

Many Bold controls support the creation and deletion of objects, for example the BoldNavigator used in the example application. Other methods exist to work with object creation and deletion as well. By using Delphi Code you can add special creation code to class operations or derived attributes, make use of the code in general GUI code, like buttons OnClick events. Bold for Delphi/Bold for C++ also support options within the model itself to help manage an objects lifecycle.

Using Delphi Code

Add two more menu items to the employee grid's popup menu with the captions 'Add New Employee' and 'Delete Employee'.

Enhance the previously created Context Popup event of the Employee grid to include the following:

```
procedure TForm1.EmployeeGridContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
  // Enable/Disable Menu items depending if an Employee is
  // selected.
  MakeManager1.Enabled := Assigned(EmployeeGrid.CurrentBoldElement);
  DeleteEmployee1.Enabled :=
    Assigned(EmployeeGrid.CurrentBoldElement);

  // Enable/Disable Menu items depending if a department is selected.
  AddNewEmployee1.Enabled :=
    Assigned(DepartmentGrid.CurrentBoldElement);
end;
```

Add on *OnClick* event handler for the 'Add New Employee' menu item:

```
procedure TForm1.AddNewEmployee1Click(Sender: TObject);
var NewEmployee: TEmployee;
    CurrentDepartment: TDepartment;
begin
  // Check that the object is a Department object and
  // assigned it to our local variable
  if DepartmentGrid.CurrentBoldElement is TDepartment then
    begin
      CurrentDepartment :=
        TDepartment(DepartmentGrid.CurrentBoldElement);

      // Create a new Employee object in the default model (we are
      // only working with one model)
      NewEmployee := TEmployee.Create(CurrentDepartment.BoldSystem);
      // Set the department of the new employee to the current
      // department
      NewEmployee.worksFor := CurrentDepartment;
    end;
end;
```

The key concept from this code is the ease of creating a new object; just create an instance of the class like any other Delphi object. Of course then you can proceed to use that object however you like.

Deleting an object is just as easy; add the following code to the *OnClick* event of the 'Delete Employee' menu item:

```
procedure TForm1.DeleteEmployeeClick(Sender: TObject);
var CurrentEmployee: TEmployee;
begin
    // Check that the object is an Employee object and
    // Delete it
    if EmployeeGrid.CurrentBoldElement is TEmployee then
        TEmployee(EmployeeGrid.CurrentBoldElement).Delete;
end;
```

Model driven object life management

Some common operations in Bold for Delphi/Bold for C++ support automatic maintenance of an objects existence.

One of the properties of an association's role is *Delete Action*. This allows Bold to control how objects in the relationship are handled when the object is deleted. The following options are available:

- **Allow**: The object can be removed. In this case the links will be severed.
- **Prohibit**: `TBoldObject.CanDelete` will return false. An attempt to delete the object anyway will raise an exception.
- **Cascade**: The related objects will be deleted as well.
- **<Default>**: See below

The default depends on the value of the *Aggregation* property:

- **none**: Allow
- **aggregate**: Prohibit
- **composite**: Cascade

We will now enhance our model to allow *Departments* to delete all *Employees* it has, when it is deleted. Likewise a *Company* will delete all *Departments* when it is deleted. Change the *Delete Action* property for the *Business Unit*, *departments* role and the *Employment*, *employs* role to *Cascade*.

Role: Company.departments -> Department

Name	departments	<input checked="" type="checkbox"/> Navigable
Stereotype		<input checked="" type="checkbox"/> Multi
Class	Department	<input type="checkbox"/> Ordered
Multiplicity	0..*	<input type="checkbox"/> Mandatory
Aggregation	none	<input type="checkbox"/> Embed
Visibility	public	
Changeability	changeable	
Delete action	Cascade	

Figure 11: Business Unit, departments role

Role: Department.employs -> Employee

Name	employs	<input checked="" type="checkbox"/> Navigable
Stereotype		<input checked="" type="checkbox"/> Multi
Class	Employee	<input type="checkbox"/> Ordered
Multiplicity	0..*	<input type="checkbox"/> Mandatory
Aggregation	none	<input type="checkbox"/> Embed
Visibility	public	
Changeability	changeable	
Delete action	Cascade	

Figure 12: Employment, employs role

Now the model will automatically cleanup the designated objects.

Run the application and checkout the change. Previously when deleting a department, the employees would still remain (visible in the All Object grid). Now they are correctly cleaned up.

Derived & Reverse Derived Attributes

So far we have only use simple attributes within our application. Each attribute has been a basic type (string & currency) and saved/restored from the database. Similar to calculated fields in a traditional Delphi database application, in Bold for Delphi/Bold for C++ it is possible to calculate the value of an attribute 'on the fly'. I say similar because Bold for Delphi/Bold for C++ extends the calculate field concept far beyond the capabilities of traditional Delphi database applications.

The most significant difference is the ability to make a derived attribute write-able. This is called reverse derived. The process of enabling this functionality is simple in a Bold application.

Adding the new attributes

The new model will include a derived attribute on the *Workplace* class and a reverse-derived attribute on the *Employee* class. The *Workplace* class now includes the *monthlyCost* attribute, which is then inherited and overridden, in both the *Company* and *Department* classes. The *Employee* class now includes the attribute *fullName* which, when changed, can make the appropriate changes to the *firstName* and *lastName* attributes.

Figure 13 shows the updated UML model.

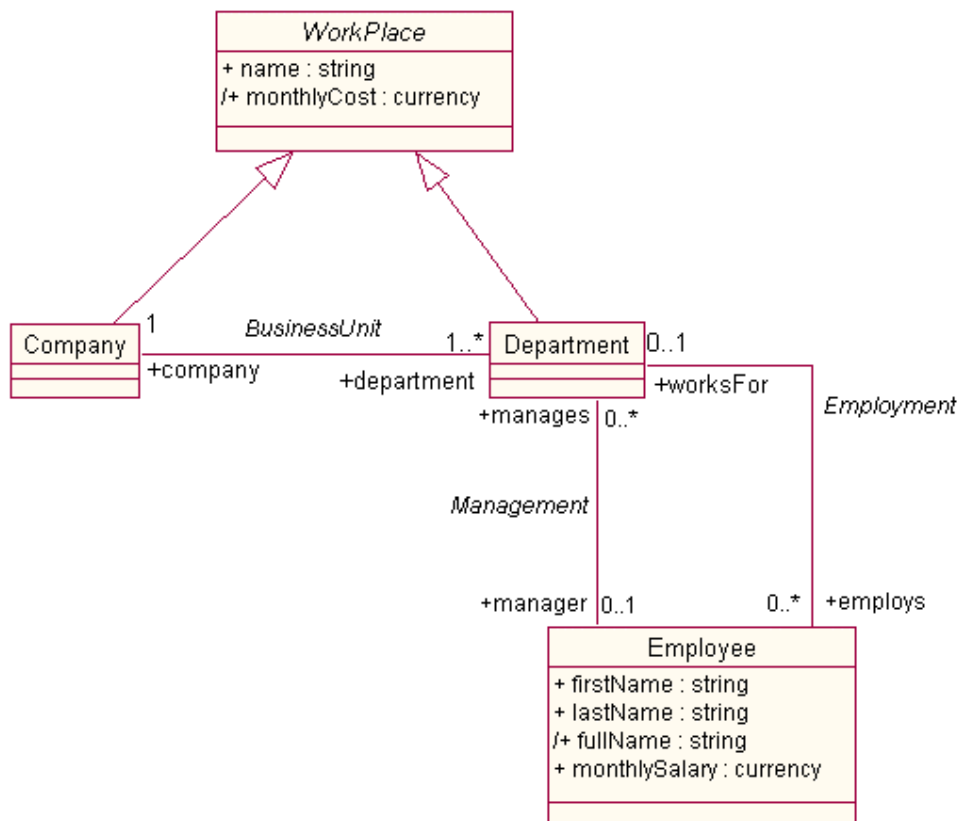


Figure 13: UML Model including new attributes

Figure 14 shows the new attributes in the Bold Model editor.

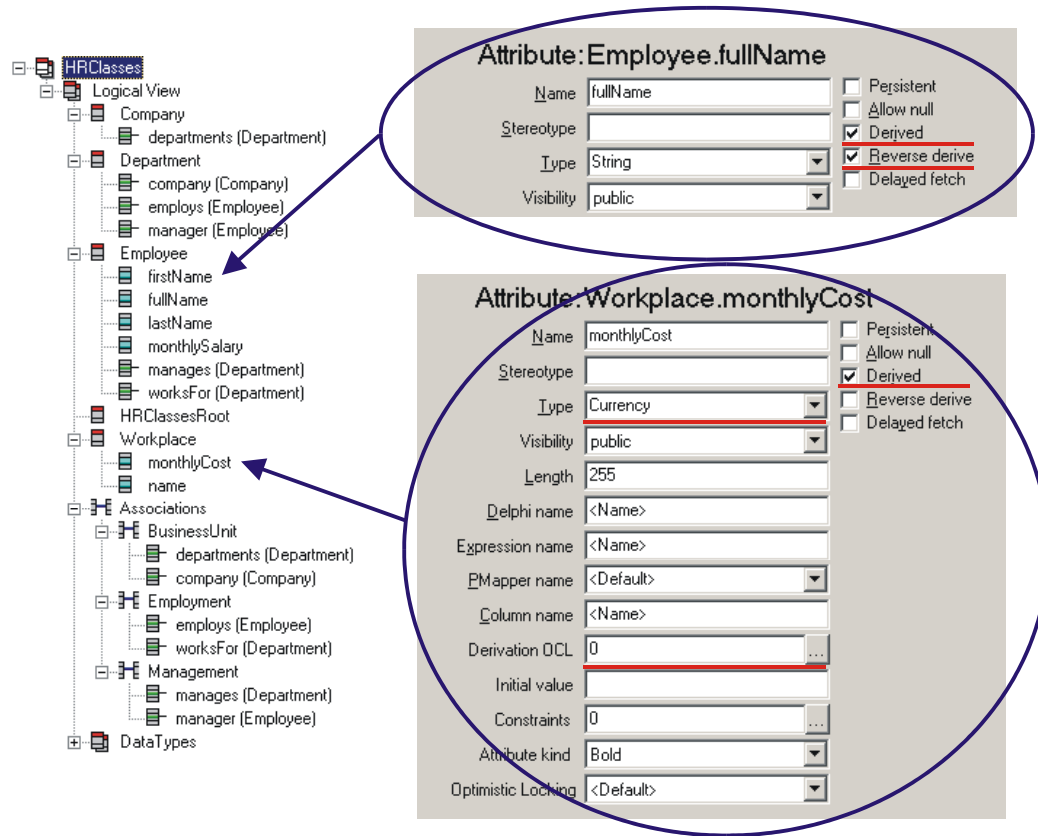


Figure 14: Adding the new attributes

After adding the new attributes it is important to use the **Generate Code** menu option from the Bold Editor to ensure the correct procedure stubs are generated. After generating the code a new HRClasses.inc file is created and loaded in the IDE editor.

The following method stubs are created for deriving and reverse deriving the employee's fullname:

```
procedure TEmployee._fullName_DeriveAndSubscribe(DerivedObject: TObject; Subscriber: TBoldSubscriber);
```

```
procedure TEmployee._fullName_ReverseDerive(DerivedObject: TObject);
```

Methods are not created for the derived *monthlyCost* attribute because we supplied an OCL expression. The expression returns a default value of zero; this will be overridden in each descendant class later.

Reversed Derived

Add the following code for the employee's *fullName* attribute:

```
procedure TEmployee._fullName_DeriveAndSubscribe(DerivedObject: TDerivedObject; Subscriber: TBoldSubscriber);  
begin  
    // Set the fullname  
    M_FullName.AsString := firstName + ' ' + lastName;  
  
    // subscribe to notifications of either the first  
    // or last name changing  
    M_FirstName.DefaultSubscribe(subscriber);  
    M_LastName.DefaultSubscribe(subscriber);  
end;  
  
procedure TEmployee._fullName_ReverseDerive(DerivedObject: TDerivedObject);  
var aFullName: String;  
    p: integer;  
begin  
    // strip away leading and trailing spaces  
    aFullName := trim(fullName);  
    // Check if a space was found  
    p := pos( ' ', aFullName );  
    if p <> 0 then  
        begin  
            // the first name is everything up to the first space  
            // the last name is the rest  
            firstName := copy( aFullName, 1, p-1 );  
            lastName := trim(copy(aFullName, p+1, maxint ));  
        end else  
            begin  
                // No space found, the first name is everything,  
                // the last name is set blank  
                firstName := aFullName;  
                lastName := '';  
            end;  
    end;  
end;
```

To understand what these methods are doing it is helpful to have a look at the declaration for the TEmployee class. The following has been stripped down to include on properties and methods important to the description of the above methods.

```
TEmployee = class(THRClassesRoot)  
    protected  
        procedure _fullName_DeriveAndSubscribe(DerivedObject: TDerivedObject;  
            Subscriber: TBoldSubscriber); virtual;  
        procedure _fullName_ReverseDerive(DerivedObject: TDerivedObject);  
            virtual;  
    public  
        property M_firstName: TBAStrng read _Get_M_firstName;  
        property M_lastName: TBAStrng read _Get_M_lastName;  
        property M_fullName: TBAStrng read _Get_M_fullName;  
        property firstName: String read _GetfirstName write _SetfirstName;  
        property lastName: String read _GetlastName write _SetlastName;  
        property fullName: String read _GetfullName write _SetfullName;  
end;
```

From this you can clearly see the methods we have just entered (in the protected section). However, it's interesting to see that we seem to have a double up of properties for each attribute.

For each property the Bold framework creates, it also creates the same property with a *M_* prefix. This second property returns a special Bold object. This object is used to represent the base type of the property, as an object. The reasons Bold for Delphi/Bold for C++ creates these runtime information classes are many:

- Allow for subscriptions to allow notifications when the value changes.
- Cache old value so changes can be discarded without refreshing from the database.
- Allow for fine-grained control of persistence at the attribute level.

If first we look at the method declaration:

```
procedure TEmployee._fullName_DeriveAndSubscribe(DerivedObject: TObject; Subscriber: TBoldSubscriber);
```

Here we have to parameters, *DerivedObject* and *Subscriber*. The *DerivedObject* in this case is the internal *TBAString* object used to represent the *fullName* attribute. This is the same object return form the *M_FullName* property. In the code we have directly access the *M_FullName* as this is more readable, we could have typecast the *DerivedObject* object as a *TBAString* and worked with that if we chose. The *Subscriber* is the internal subscription object for this property, it is our responsibility to use this object to subscribe to any other object attributes that we need to know have changed. This subscription allows for the *fullName* property to respond to the change of *firstName* or *lastName* and correctly show the updated value.

Setting the property value:

```
M_FullName.AsString := firstName + ' ' + lastName;
```

The employee's full name is simply the first name and last name separated with a space.

```
M_FirstName.DefaultSubscribe(subscriber);  
M_LastName.DefaultSubscribe(subscriber);
```

Here the subscriber for the *fullName* attribute attaches itself to the subscription mechanism for each of the other name attributes.

If you run the application and use the default object form, you can see the read/write behavior of the new *fullName* attribute in action.

Derived Attributes with OCL

In the descendant class of *Workplace* we can override the OCL for the *monthlyCost* attribute. This allows for the full use of polymorphism by changing the behavior of the attribute for each descendant class.

To override OCL derived attributes in Bold for Delphi/Bold for C++ place an entry in the *Derived Expressions* property of the class. This is shown in Figure 15.

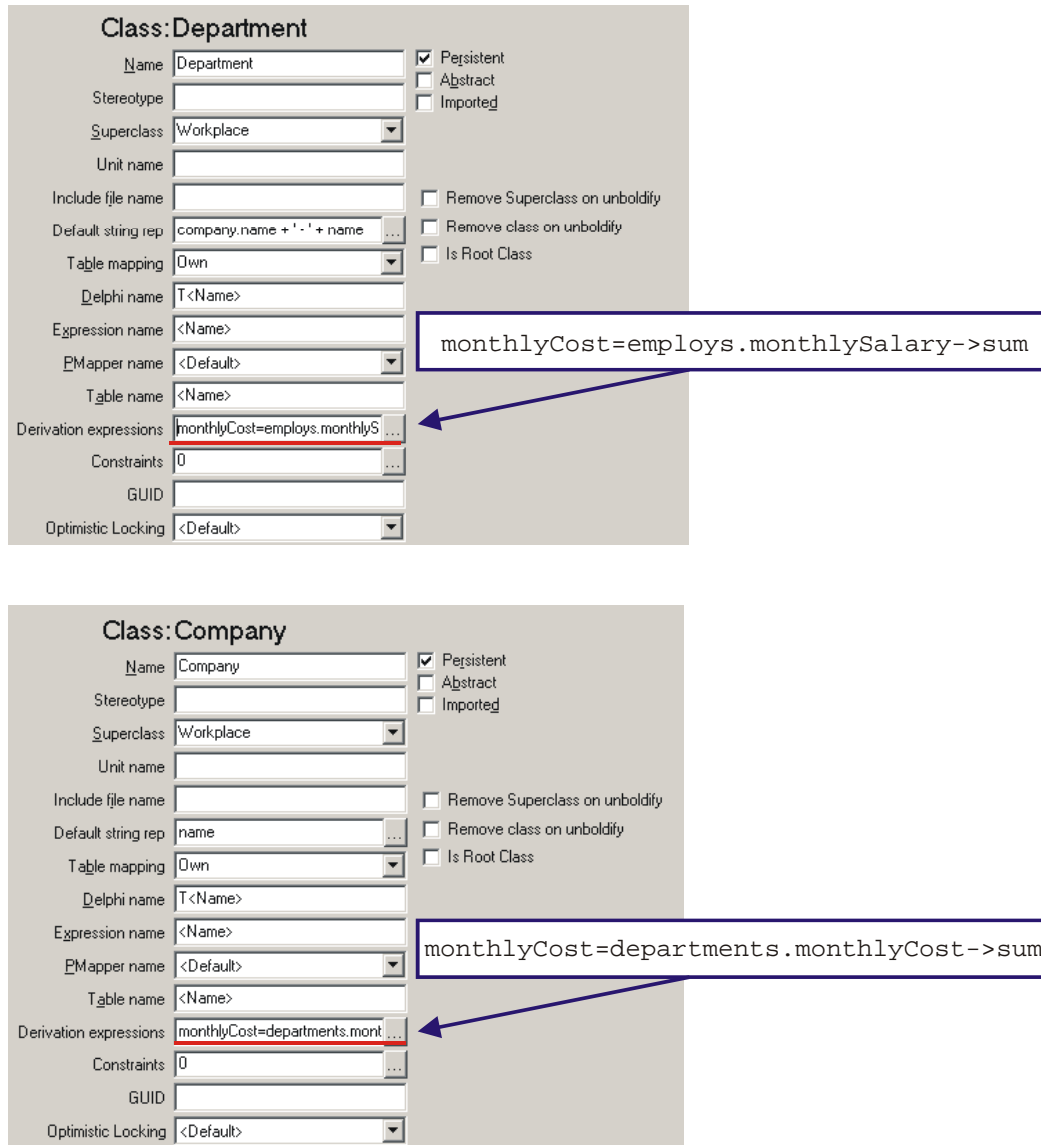


Figure 15: Overriding derived OCL expressions

The *Derivation Expressions* property of the class contains a string list of all the overridden derived OCL expressions. The format of each line is:

```
<inherited attribute name>=<new OCL expression>
```

In the example above the *Department* class now calculates the *monthlyCost* value as being the sum of all employee's salaries for that department. The *Company* class calculates the *monthlyCost* value as being the sum of all monthly costs for each department.

For both the Company and the Department grid we can add a new column and set the OCL expression to *monthlyCost*. At runtime the screen will look similar to Figure 16.

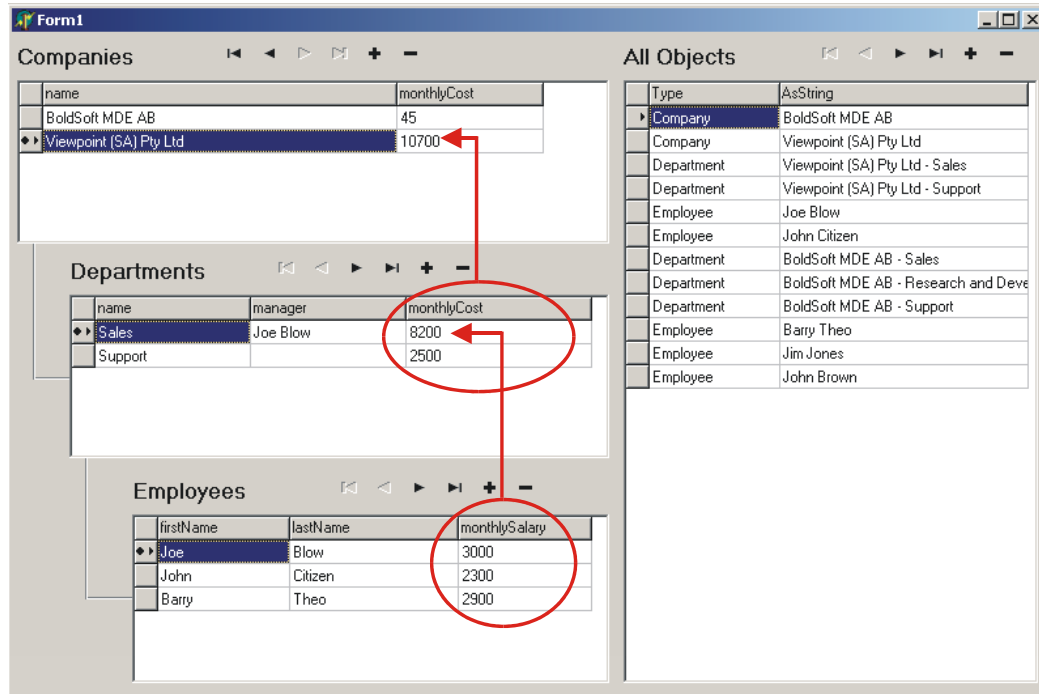


Figure 16: Derived Attributes at Runtime

Operations

So far we have discussed attributes and relationships, another very important part of Bold for Delphi/Bold for C++ is *Operations*. Operations allow you to add behavior to your classes and are vital to allowing you to implement all the required business rules for an application.

Operations are implemented using standard Delphi Object Pascal. They are the equivalent to methods in standard Delphi classes. Operations are used to implement tasks within your classes, like *LightBulb.change*.

Adding an Operation

Add the *adjustSalary* operation shown in the updated UML model in Figure 17.

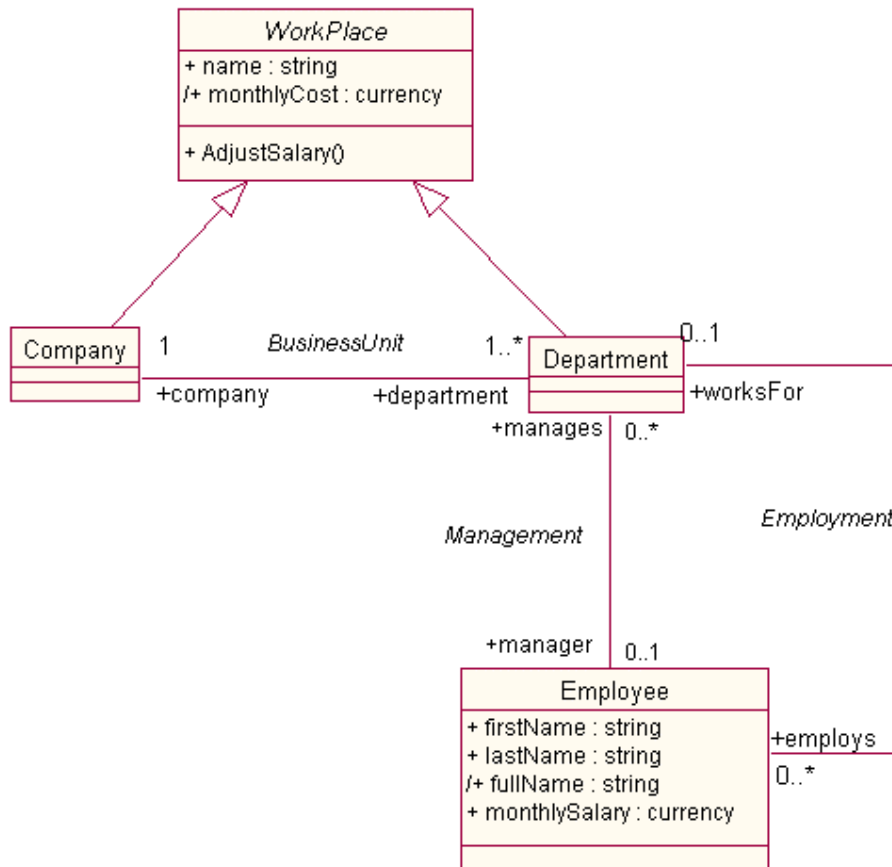


Figure 17: Adding an Operation

The *adjustSalary* operation doesn't need to do anything in the base *Workplace* class. In both the *Company* and *Department* classes it will be overridden to supply different behavior for each class.

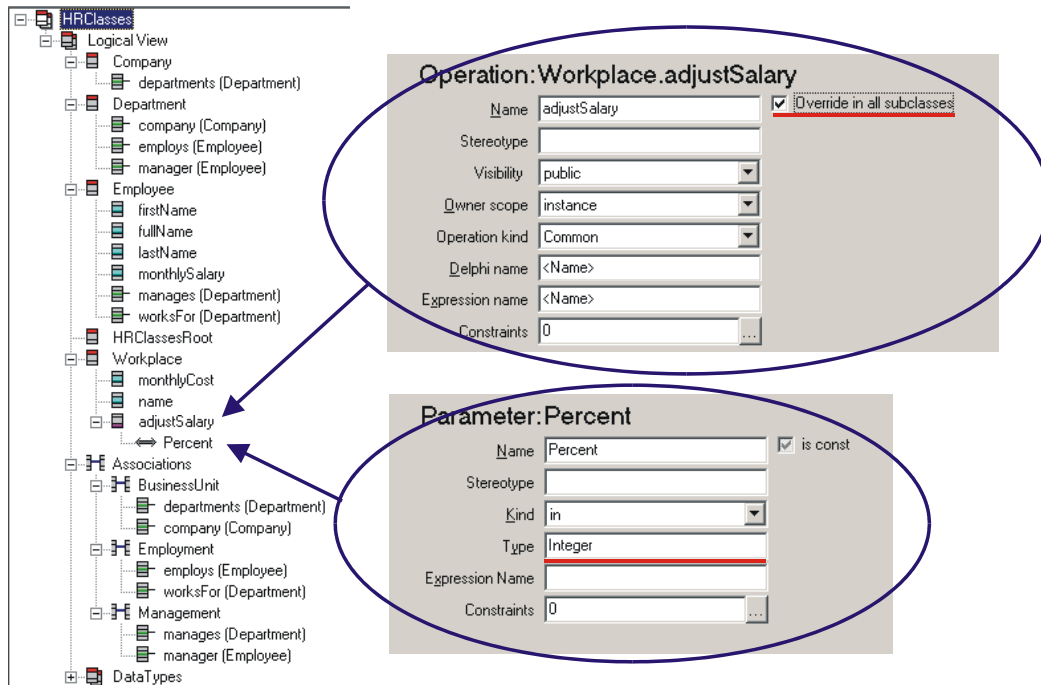


Figure 18: Adding the adjustSalary Operation

After adding the operation you will need to use *Generate Code* from the *Tools* menu. The following method stubs are added to *HRClasses.inc*.

```
procedure TWorkplace.adjustSalary(Percent: Integer);  
begin  
end;
```

```
procedure TCompany.adjustSalary(Percent: Integer);  
begin  
  inherited;  
end;
```

```
procedure TDepartment.adjustSalary(Percent: Integer);  
begin  
  inherited;  
end;
```

No behavior is required in the *Workplace* class so we will only add code to the *Company* and *Department* implementations.

```
procedure TCompany.adjustSalary(Percent: Integer);  
var counter: Integer;  
begin  
  inherited;  
  // Loop thru all departments and call  
  // adjustSalary for each one  
  for Counter := 0 to departments.Count -1 do  
    departments[counter].adjustSalary(Percent);  
end;
```

```
procedure TDepartment.adjustSalary(Percent: Integer);  
var counter: Integer;  
    SalaryChange: Currency;  
begin  
    inherited;  
    // Loop thru all employee's and adjust their  
    // salary  
    for Counter := 0 to employs.Count -1 do  
    begin  
        SalaryChange := employs[counter].monthlySalary * (Percent / 100);  
        employs[counter].monthlySalary := employs[counter].monthlySalary  
            + SalaryChange;  
    end;  
end;
```

To show the effect of this new operation we will add a pop-up menu to both grids and allow the user to adjust salaries.

Add a new TpopupMenu to the form and call it SalaryPopupMenu. Set both the CompanyGrid and the DepartmentGrid to use this new popupmenu by setting the PopupMenu parameter of each grid.

Add a private field to the form's class declaration:

```
type  
    TForm1 = class(TForm)  
    ...  
    private  
        { Private declarations }  
        FCurrentWorkplace: TWorkplace;  
    public  
        { Public declarations }  
    end;
```

Add the following code to the OnContextPopup event of the DepartmentGrid:

```
procedure TForm1.DepartmentGridContextPopup(Sender: TObject;  
    MousePos: TPoint; var Handled: Boolean);  
begin  
    // Initialize menu item and form private field  
    AdjustSalary1.Enabled := False;  
    FCurrentWorkplace := nil;  
  
    // Check that the object is a Workplace object and  
    // assigned it to our local variable  
    if DepartmentGrid.CurrentBoldElement is TWorkplace then  
    begin  
        AdjustSalary1.Enabled := True;  
        FCurrentWorkplace :=  
            TWorkplace(DepartmentGrid.CurrentBoldElement);  
    end;  
end;
```

Add the following code to the OnContextPopup event of the CompanyGrid:

```
procedure TForm1.CompanyGridContextPopup(Sender: TObject; MousePos:
TPoint; var Handled: Boolean);
begin
    // Initialize menu item and form private field
    AdjustSalary1.Enabled := False;
    FCurrentWorkplace := nil;
    // Check that the object is a Workplace object and
    // assigned it to our local variable
    if CompanyGrid.CurrentBoldElement is TWorkplace then
        begin
            AdjustSalary1.Enabled := True;
            FCurrentWorkplace :=
                TWorkplace(CompanyGrid.CurrentBoldElement);
        end;
end;
```

Both of these events are very similar. They check that a valid *Workplace* is selected in the grid and assign it to the local form's private field *FcurrentWorkplace*. The popup menu can then use this.

Add a menu item to the popup menu with the caption 'Adjust Salary'. In the menu items *OnClick* event add the following code:

```
procedure TForm1.AdjustSalary1Click(Sender: TObject);
var PercentStr: String;
    PercentInt: Integer;
begin
    // Ensure that the local form field is assigned
    // an object
    if assigned(FCurrentWorkplace) then
        begin
            // Display a pop-up requesting the percent to
            // adjust the salaries by
            PercentStr := '10';
            if InputQuery('Salary Adjustment', 'Enter adjustment percentage',
                PercentStr) then
                begin
                    try
                        // call the adjustSalaries method of with the supplied percent
                        PercentInt := StrToInt(PercentStr);
                        FCurrentWorkplace.adjustSalary(PercentInt);
                    except
                        on EConvertError do
                            MessageDlg('Invalid Percent entered!', mtError, [mbOK], 0);
                    end;
                end;
        end;
end;
```

The above code works for both grids because we use the base class of the objects making the code type compatible. This is an enormous benefit in reducing redundant code and helps ensure code re-use. The method simply prompts for an adjustment percentage and calls the *adjustSalary* method of the object. If the object is a *Company* then the *adjustSalary* method of that class is used, if the object is a *Department* then that version of the method is used. Fortunately that fact doesn't need to be known by the popup code above, Object Pascal or C++ and the Bold framework take care of this for us.

Derived Relationships

With the power of derived, reverse derived attributes and operations it is easy to see how extending a Bold for Delphi/Bold for C++ application to truly integrate business rules into the model is easy. Bold for Delphi/Bold for C++ can extend this even further by deriving relationships 'on the fly'. The advantage of this is, as previously discussed, is reduced complexity, centralized business rules, ease of maintenance. This also reduces redundancy of data, sometime when implementing a database the rules of normalization are broken to accommodate reports or programming requirements. Data that can be easily derived from existing information is instead duplicated in one or more tables. This can result in problems as the application evolves or problems during normal operation ensuring the data is kept synchronized.

By using derived relationships, the benefit of having the information readily available and without redundant data can help ease the application development and maintenance. Figure 19 shows two new relationships in our model, *Workforce* and *Managed By*.

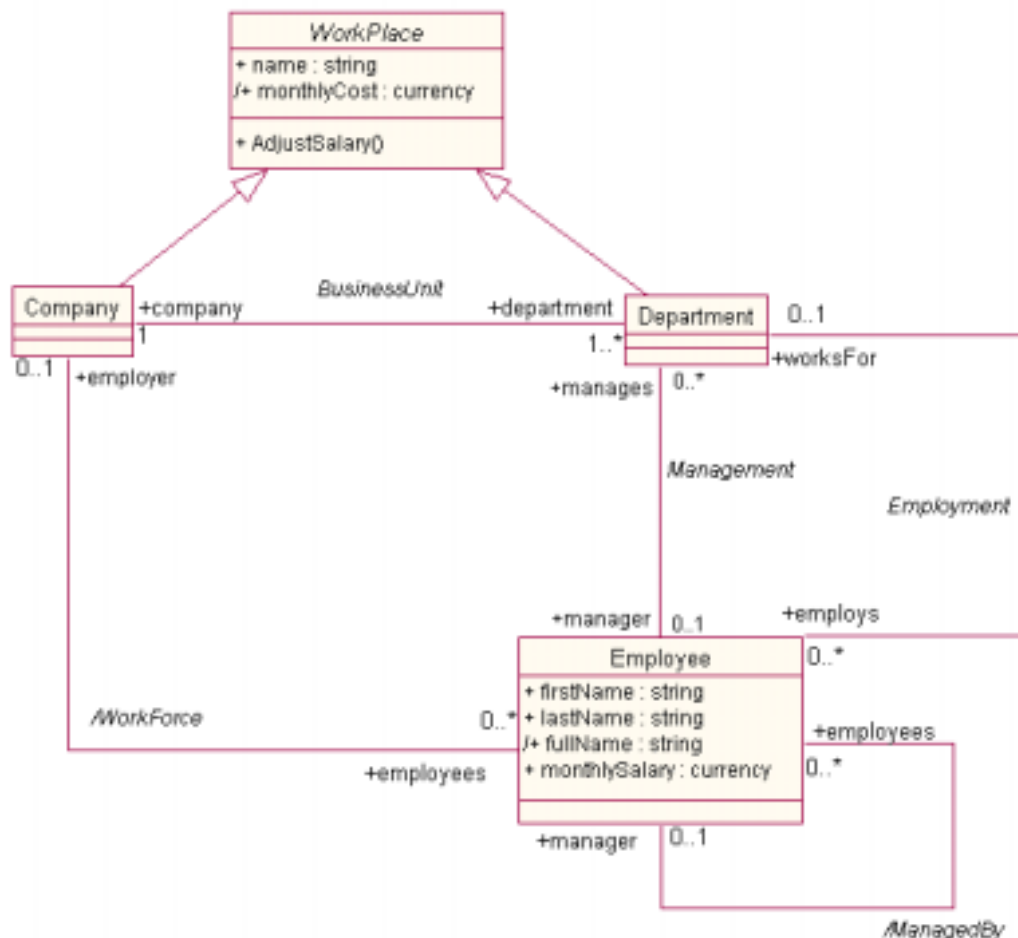


Figure 19: Derived Relationships

The *Managed By* relationship represents the manager for that employee; this is simply whoever is managing the department the employee works for. The *Workforce* relationship represents the employer of the employee; this is the company that owns the department that the employee works for.

Adding the *Managed By* relationship

Add the *Managed By* relationship as per Figure 20:

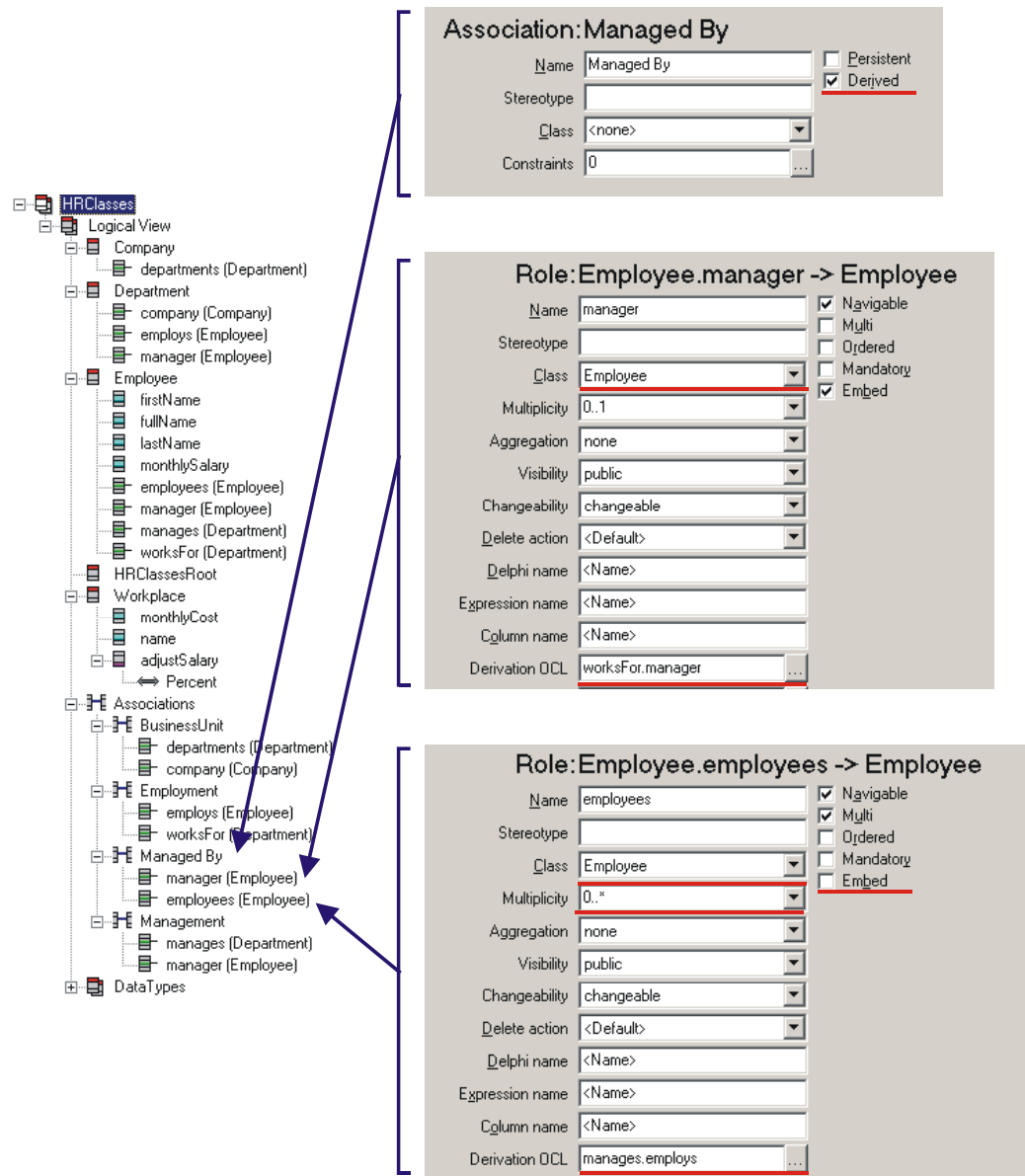


Figure 20: Derived Relationship – Managed By

The OCL expression `worksFor.manager` will return the employee who manages the department the current employee works for. The OCL expression `manages.employs` will return a list of employees. This list is the combination of all employees that work for all the departments that the current employee manages.

This behavior can be observed at runtime using the Bold for Delphi/Bold for C++ object forms. Open an employee and drag a few departments into the *manages* tab. The *employees*' tab now lists all employees for the departments managed by that employee.

Adding the Workforce relationship

Add the *Workforce* relationship as per Figure 21:

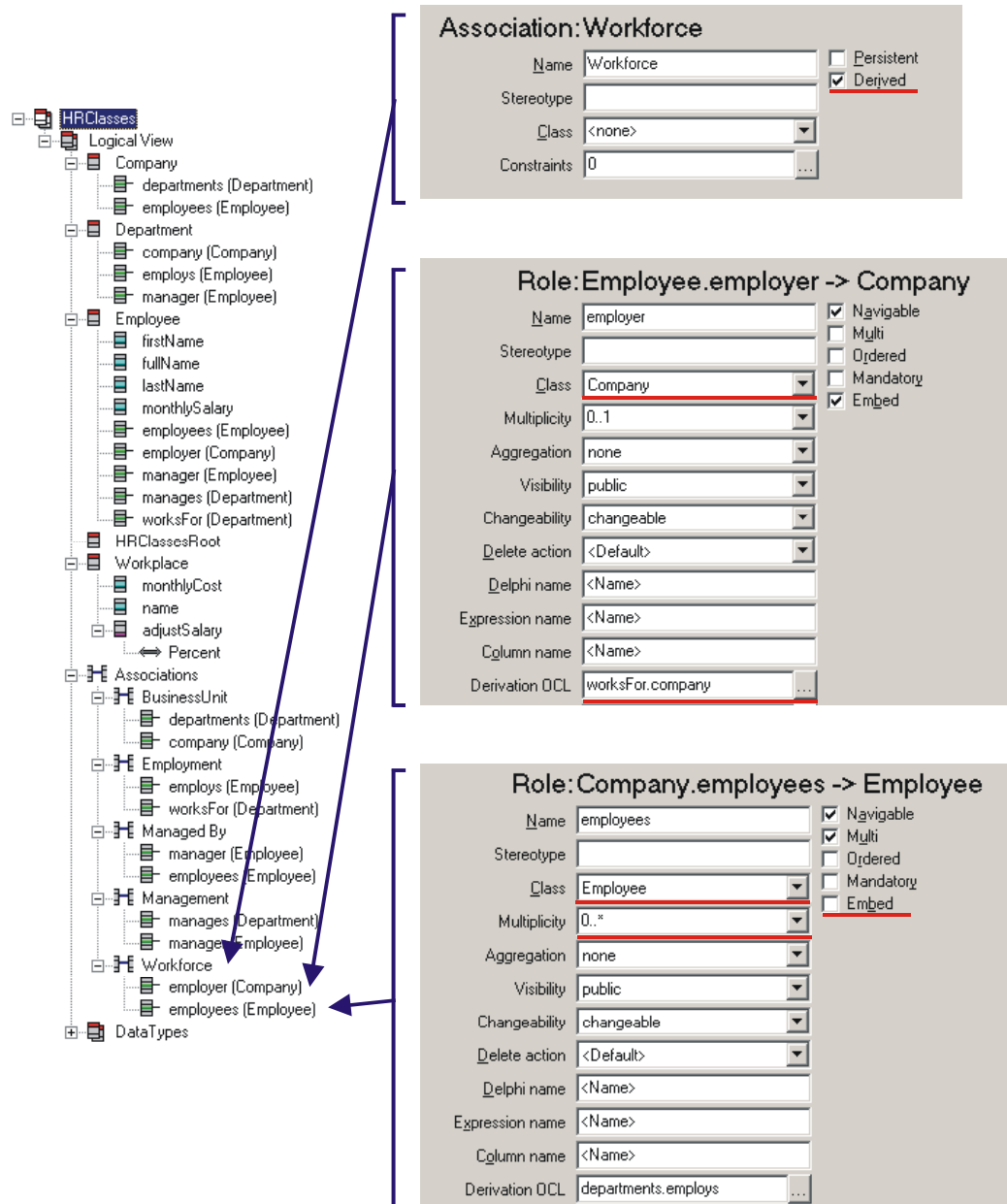


Figure 21: Derived Relationship – Workforce

The behavior can easily be shown by using the default Bold for Delphi/Bold for C++ object forms at runtime. The new *employee* tab for the *Company* class shows all employees for all departments. The *Employer* field for the *Employee* class now shows the employer.

One-way Relationships

We will now take a look at a one-way relationship, although not too different from what we have already done, it will give us an opportunity to explore some more of the options available with relationships.

In the UML model in Figure 22 the relationship *Top Salaries* has been added. This provided an association from *Workplace* to *Employee* identifying *highlyPaidEmployees*. To help facilitate the use of this new relationship the attribute *highSalaryThreshold* has been added to the *Department* class.

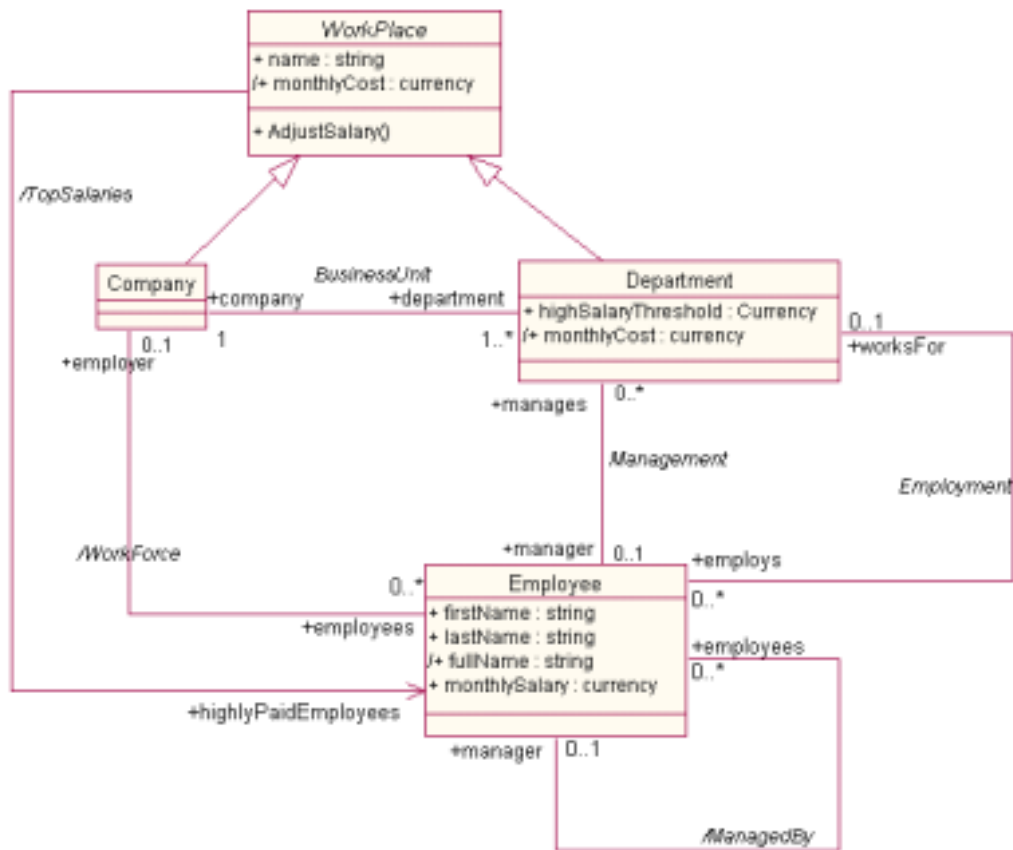


Figure 22: One-way Relationship

This is implemented in the Bold Model Editor as follows:

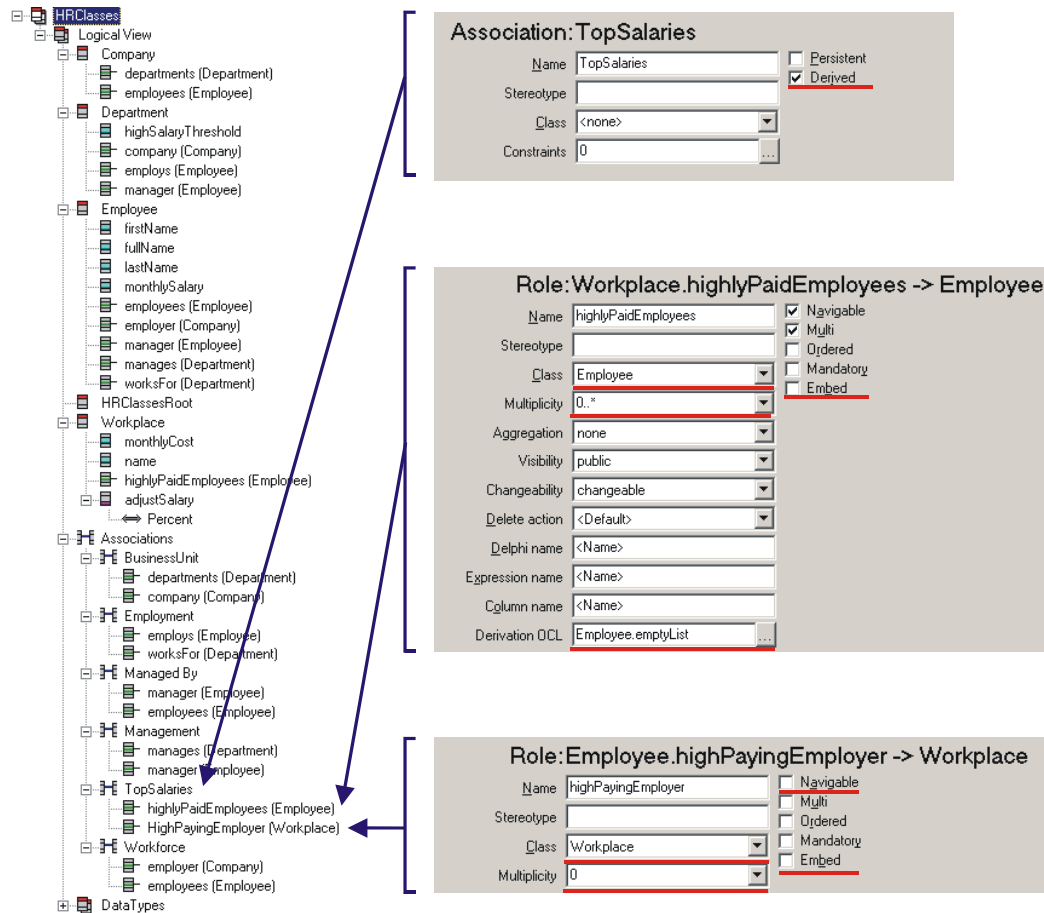


Figure 23: Implementing Top Salaries

The interesting items when implementing this association are:

- 1) The role *highlyPaidEmployees* uses the OCL expression `Employee.emptyList`. This expression results in an empty list of type *Employee* objects. The reason we use this expression is because the *Class* of the association is *Employee*. Even though *Workplace* is an abstract class, all descendant classes will inherit this association. It is important that it returns a valid result if called. If the OCL expression were blank Bold for Delphi/Bold for C++ would create a method stub to allow you to create your own result in code. Because we will be overriding the result using OCL in descendants the expression above was required.
- 2) The role *highPayingEmployer* is not required in a business sense, indeed deriving a result really adds no value to the model at all. Having the role here is important because assigning the *Class* property to *Workplace* gives us the context for the other role. The trick to removing this otherwise useless role from appearing in the *Employee* class and generated code is to mark it as not *Navigable*. This was done by unchecking the *Navigable* checkbox.

At the moment the association offers no value, as it will always return an empty list. By overriding the OCL expression in descendant classes we can generate a list of *Employee*'s.

Add the following attribute to the *Department* class:

Attribute: Department.highSalaryThreshold

Name	highSalaryThreshold	<input checked="" type="checkbox"/> Persistent
Stereotype		<input type="checkbox"/> Allow null
Type	Currency	<input type="checkbox"/> Derived
Visibility	public	<input type="checkbox"/> Reverse derive
		<input type="checkbox"/> Delayed fetch

Figure 24: Attribute – highSalaryThreshold

This attribute will be used to determine which employees in a department are considered as having a high salary. This means we need to now override the implementation of *highlyPaidEmployees* in both the *Company* and *Department* classes.

This is done by adding another entry to the *Derivation Expression* property of the two classes.

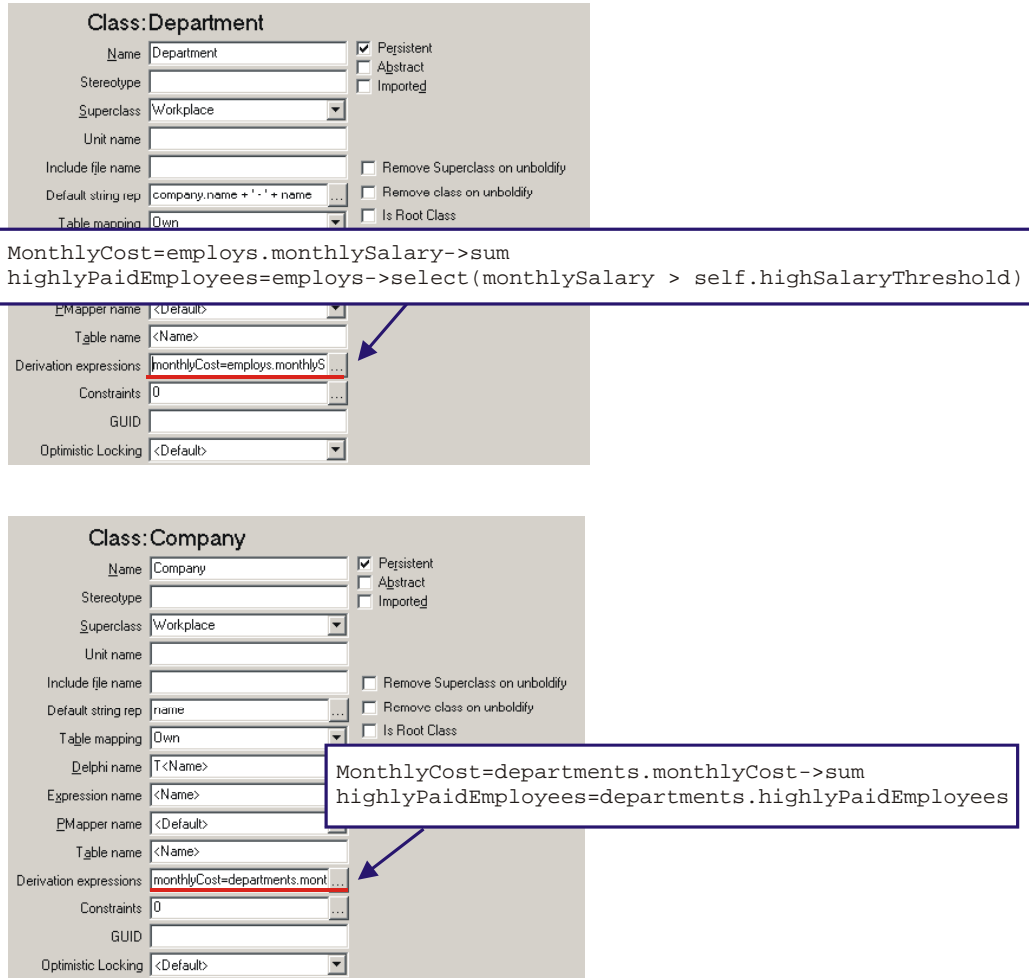


Figure 25: Derivation Expressions

The OCL expression used for the *Department* class uses the *highSalaryThreshold* attribute to determine which employees are considered as being highly paid. The *Company* class simply aggregates all the highly paid employees of its departments to provide a consolidated list.

The behavior can be investigated using the Bold for Delphi/Bold for C++ object forms at runtime, however let's add a few more controls to give us a result on our main form. Figure 26 shows the enhanced main form at design time.

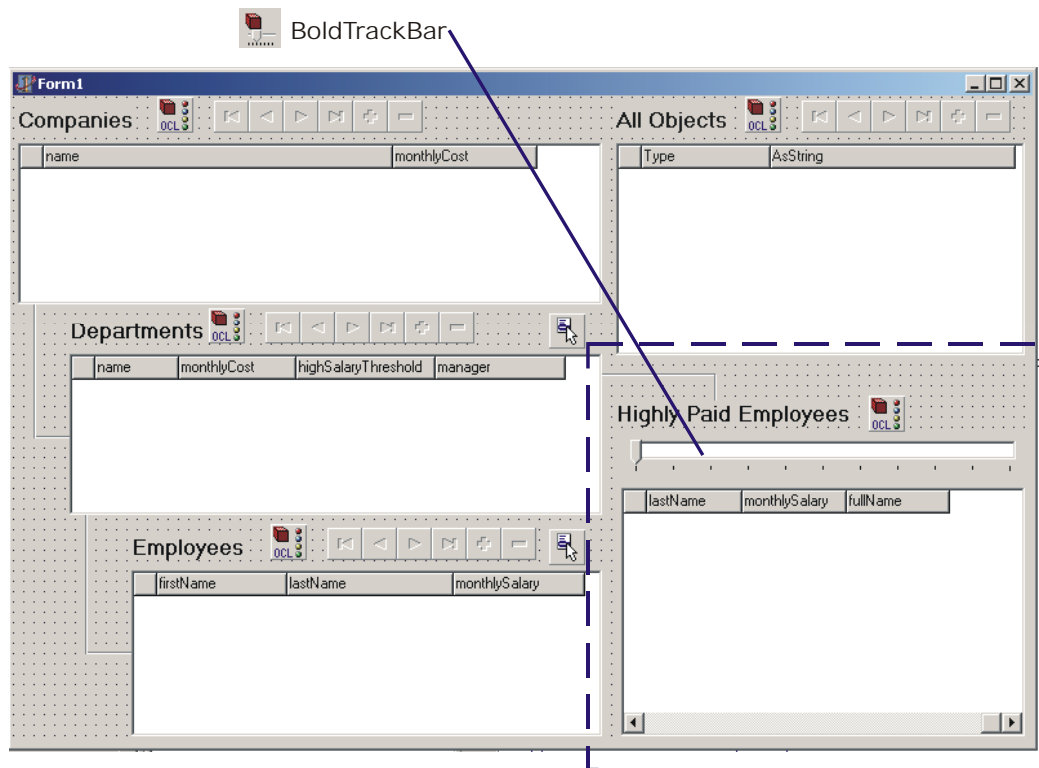


Figure 26: Enhancing the GUI

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

BoldListHandle

Name: HighlyPaidEmployeesList
RootHandle: DepartmentList
Expression: highlyPaidEmployees
Enabled: True

BoldTrackBar

Name:	SalaryThresholdTrackBar
BoldHandle:	DepartmentList
BoldProperties.Expression:	highSalaryThreshold
BoldProperties.ApplyPolicy:	bapChange
Min:	0
Max:	5000
Frequency:	500
Enabled:	True

BoldGrid

Name:	HighlyPaidEmployeesGrid
BoldHandle:	HighlyPaidEmployeesList

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

Run the application. Choose a department and add several employees with a spread of monthly salaries between 0 and 5000. Now slide the trackbar, the Bold framework dynamically responds with the correct result in the grid.

If you open the default editor for the *Company* and select the *highlyPaidEmployees* tab, you will see the highly paid employees for all departments. The beauty of the Bold framework is this list also dynamically updates as you move the slider. The synchronization methods of Bold for Delphi/Bold for C++ are extremely powerful and make creating loosely coupled forms easy.

Constraints: A parting note

An important aspect of Bold for Delphi/Bold for C++ is constraints. These are used to validate objects. Constraints can be specifically entered or implicit based on a relationship role. For example the role *Company* as part of the *BusinessUnit* association has a multiplicity of 1 specified. This means that every department must have a relationship with a company. However, as you can quickly check by running the application (prior to adding the cascade delete) that it is certainly possible to do so.

It would be to limiting or impractical for the Bold framework to try and enforce this constraint automatically as almost certainly the resulting behavior won't suit your particular scenario. Bold for Delphi/Bold for C++ however provides the ability to test the constraints, even as part of the delete verification process for your objects.

This is mentioned only because you may have been aware that certain aspects of the model were not being enforced in this way. However the Bold for Delphi/Bold for C++ constraint mechanism is powerful and fully assessable at runtime via code and Bold components.

The subject of constraints will be addressed in a future article.

Summary

A lot of information has been presented in this article. The power of moving information about the application from the GUI and source into the model allows for a more precise definition of business rules. The centralized containment of business rule is a good way to ensure they are implemented in the application.

Designing an application with the knowledge everything modeled can actually be implemented, without worrying about technical details such as DB schema, linked lists, updating queries and moving data from the GUI into the logic layer allows focus to stay on the domain problem at hand. Keeping focus on the domain problems, rather than implementation problems ensures energy is spent on the right issues.

Bold for Delphi/Bold for C++ offers the most powerful assurance ever produced that your application is actually based on the model, and that the model will always accurately reflect the true implementation.