CHAPTER 9

Introducing Business Components for Java

"Contrariwise," continued Tweedledee,
"if it was so, it might be, and if it were so,
it would be; but as it isn't, it ain't. That's logic!"

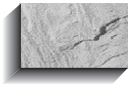
—Lewis Carroll [Charles Lutwidge Dodgson] (1832–1898),
Through the Looking Glass

usiness Components for Java (BC4J) is a Java- and XML-based framework for developing business logic, including validation and default logic, queries, transaction handling, and data access. The BC4J layer of an application does not create a user interface at all. It is a pure encapsulation of business logic that communicates with a separate client application, which handles user interaction.

Chapters 1 and 3 provided an overview of the range of features JDeveloper provides and the range of activities you can perform with it. In Part II, you will learn in depth about BC4J, one of JDeveloper's central features. Using BC4J is the simplest way to design data-aware applications with JDeveloper. However, Part II will not teach you everything you need to write a complete BC4J application. In addition to a collection of business components, BC4J applications involve client applications (sometimes called "clients" in Part II), which you will learn to create in Parts III and IV.

This chapter provides an overview of BC4J, describing the components and uses of BC4J. Chapter 10 explains how to create business components to represent database objects, object types, and constraints. Chapter 11 explains how to reverse the process and design business components in isolation from the database before creating database objects to provide a persistence layer. In Chapter 12, you will learn how to use BC4J to enforce business rules.

Chapter 13 explains how BC4J represents queries and relationships between query results and how it caches data. In it you will learn how to assemble the data your client needs while maintaining optimal performance. Chapter 14 explains how clients access the data and presents techniques for accessing data that will help you customize the client applications you will learn about in Parts III and IV. In Chapter 15, you will learn how to use the BC4J framework to customize transaction handling and scale your applications to effectively handle a large number of simultaneous transactions. In Chapter 16, you will learn more about deploying BC4J applications and the factors you should weigh in deciding how to deploy your business components.



NOTE
Many practices in Part II are sequential, but you can download starting files for any chapter's practice from the authors' websites mentioned in the front of the book.

This chapter includes a brief discussion of the advantages and disadvantages of using BC4J to enforce business logic and a quick tour of the different business components: entity objects, associations, view objects, view links, application modules, and domains. At the end of the chapter, a hands-on practice will demonstrate how to create and explore a simple business components application.

Why Use Business Components for Java?

Why create a new application layer to enforce business logic, rather than enforcing it in the user interface or in the database itself?

The advantage of BC4J over UI-enforced business logic is reusability. A single BC4J layer can provide the business logic for all of your company's needs. The business components can be used

again and again, with multiple user interfaces. This saves you from having to rewrite business logic for every user interface.

The advantages of BC4J over database-enforced business logic are more complex. First, by maintaining a cache of data in memory, BC4J reduces the number of database trips required by your application, thus improving performance and scalability. Second, BC4J allows you to write your business logic in Java, saving you the trouble of integrating a Java GUI or JSP application with business logic written in PL/SQL code. Finally, moving business logic out of the database keeps the database from handling anything but data, which increases your application's modularity and the database's efficiency.

There are also advantages to implementing some business logic directly in the database. Business logic in the database is more robust. It will be enforced even in a SQL*Plus session. Business logic implemented in a BC4J layer will only be enforced in applications using that layer. This trade-off between robustness and performance is something you need to consider for your application. The most critical business logic should be implemented in the database or implemented redundantly in the database and a BC4J layer. The remainder of your business logic can be implemented in the BC4J layer alone.

Although you can write your own client code for BC4J, JDeveloper provides four ready-made architectures for business component clients:

- JClient, an architecture for creating Java GUIs (especially useful for intensive data-entry applications that require high interactivity) discussed in Part III of this book
- Thin JSP clients (useful for web-based applications for which a thin client is more important than very high interactivity, such as self-service applications and e-commerce applications) discussed in Part IV of this book
- XSQL clients, which generate XML for either flexible display with a style sheet or loose coupling between applications
- UIX, a framework-based architecture for developing web applications

XSQL clients and UIX are beyond the scope of this book.

BC4J also has the advantage of being "deployment-configuration independent." A single BC4J layer can be deployed as local Java classes or as an Enterprise JavaBean (EJB), with no rewriting of code necessary. Programmers using business components, unlike those creating Enterprise JavaBeans from scratch, do not need to design applications around their deployment architecture; they can concentrate on their business logic rather than on deployment requirements. You will learn more about maintaining tier independence in Chapter 14 and about how to deploy BC4J in various configurations in Chapter 16.

Whether deployed to a J2EE web module with JSP pages or as an EJB, business components are fully J2EE-compliant and, in fact, the BC4J framework automatically implements the J2EE BluePrints design patterns suggested by Sun Microsystems. You do not need to worry about these design patterns (Sun discusses them extensively on their website, java.sun.com); if you create a business components application, you will automatically be designing to them.

Entity Objects and Associations

An *entity object* usually represents a database table or database view, although it can also be used to represent EJB entity beans (for more information, see the sidebar "EJB Entity Facades"). It acts as a representation of the table or database view and handles all of the business rules for that

EJB Entity Facades

Until recently, BC4J entity objects could only be used to wrap database objects directly. In release 9.0.3, however, JDeveloper introduced the ability to use special types of entity objects, called EJB entity facades, as wrappers for EJB entity beans. In this configuration, the EJB entity beans handle data persistence; the entity facades handle business logic and (unlike standard entity objects) client binding.

Entity facades work a bit differently from standard entity objects, and if you use them, you will also use new forms of other business components (such as a new form of a view object called an EJB finder view object). A discussion of entity facades and related business components is beyond the scope of this book. For more information, see the on-line help.

Note that, although entity facades and related business components are fully J2EE compliant, standard business components are as well. You do not need to use entity facades unless you have a special reason to use EJB entity beans.

table or view including validation, defaulting, and anything else that happens when a row is created, deleted, or changed.

The power of BC4J is its interface with the database to be used in an application. A relational database consists of a set of tables, each of which contains columns. For example, consider the table DEPARTMENTS, with the following columns:

Column Name	SQL Datatype
DEPARTMENT_ID	NUMBER
DEPARTMENT_NAME	VARCHAR2
MANAGER_ID	NUMBER
LOCATION_ID	NUMBER

The BC4J layer represents the database table as an entity object. An entity object has *entity attributes*, which represent the table columns, although the mapping is not always perfectly one-to-one. (For more information, see Chapter 10.)

The types of the properties are Java classes that correspond to the SQL types of the columns. The entity object for this table might have the following entity attributes:

Attribute Name	Java Type
DepartmentId	oracle.jbo.domain.Number
DepartmentName	java.lang.String
ManagerId	oracle.jbo.domain.Number
LocationId	oracle.jbo.domain.Number

Java does not directly support SQL datatypes. However, each SQL datatype can be mapped to a Java type. Some of these Java types are classes in java.lang (such as java.lang.String), and others are in the package oracle.jbo.domain (which are discussed in the section "Domains").

An entity object has two parts: a Java class (such as DepartmentsImpl.java) and an XML file (such as Departments.xml). The Java class contains procedural code required to implement the entity object, while the XML file includes metadata describing the entity object, its attributes, and the table upon which it is based, as well as declarative code. These will be examined in more detail later in this chapter.

Associations

Just as tables are often related to one another, entity objects are often related to one another. Relationships between entity objects are represented by associations. You can think of an *association* as the representation of a foreign key relationship: it matches one or more attributes of a "source" entity object with one or more attributes of a "destination" entity object, just as a foreign key constraint matches one or more columns of a child table with one or more columns of a parent table. You can (but are not required to) base an association on a foreign key constraint in the database. The association is stored in an XML file.

You will learn how to define entity objects and associations in Chapters 10 and 11 and how to use them to implement business logic in Chapter 12.

View Objects and View Links

An entity object usually represents a table or view in the database. But you generally should not present all of the information stored in a database object in one application interface. Also, you may want data taken from more than one database object. SQL has queries so that you can select exactly the data that you need from one or more tables. This is also the reason why Business Components for Java has *view objects,* which correspond to SQL queries. A view object actually stores a SQL query.

Just as an entity object has entity attributes, a view object has *view attributes,* which correspond to columns of the query result. For example, consider the view object for the following query:

```
SELECT Departments.DEPARTMENT_ID,
  Departments.DEPARTMENT_NAME,
  Employees.EMPLOYEE_ID,
  Employees.FIRST_NAME,
  Employees.LAST_NAME
FROM DEPARTMENTS Departments, EMPLOYEES Employees
WHERE Departments.MANAGER_ID=Employees.EMPLOYEE_ID;
```

This view object would have the following view attributes:

Attribute Name	Java Type
DepartmentId	oracle.jbo.domain.Number
DepartmentName	java.lang.String
EmployeeId	oracle.jbo.domain.Number
FirstName	java.lang.String
LastName	java.lang.String

These view attributes can be, but do not need to be, associated with attributes from entity objects.

Like an entity object, a view object has two parts: a Java class (such as ManagerViewImpl.java) and an XML file (such as ManagerView.xml).The Java class handles the complex logic of your queries, controlling the client's access to the data. The XML file stores information about the query and its relationships to entity objects.

View Links

A *view link* represents a relationship between the query result sets of two view objects. It associates one or more attributes of one view object with one or more attributes of another. For example, you could create the following:

- A view object, DepartmentsView, containing the following query:

```
SELECT Departments.DEPARTMENT_ID,
  Departments.DEPARTMENT_NAME
FROM DEPARTMENTS Departments
```

- Another view object, EmployeesView, containing the following query:

```
SELECT Employees.EMPLOYEE_ID,
  Employees.FIRST_NAME,
  Employees.LAST_NAME,
  Employees.DEPARTMENT_ID
FROM EMPLOYEES Employees
```

- A view link, DeptEmpFkLink, that associated the DepartmentId attribute of EmployeesView with the DepartmentId attribute of DepartmentsView

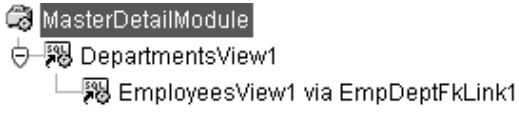
DeptEmpFkLink represents a master-detail relationship between the query result sets of DepartmentsView and EmployeesView.

View links between view objects can be, but do not need to be, based on associations between underlying entity objects. A view link is represented in an XML file.

You will learn how to design view objects and view links in Chapter 13 and how to use them in client applications in Chapter 14.

Application Modules

An *application module* is a container for *view usages*, instances of view objects. It lists all of the view usages that your application requires and specifies the way in which they are related by view links. These relationships can be represented by a tree, called the application module's *data model*. For example, an application module might contain a usage of DepartmentsView, called DepartmentsView1, and a usage of EmployeesView, called EmployeesView1, linked by a usage of the view link EmpDeptFkLink called EmpDeptFkLink1. The application module would use the data model shown here:

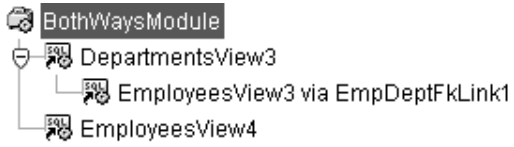


Using this model, the two view usages are tied together by a link representing a master-detail relationship between them. Through this application module, your client application could select a row in DepartmentsView1, and the BC4J framework would immediately synchronize EmployeesView1 so that it would only return employees from the selected department.

Alternatively, your application module could contain a usage of DepartmentsView and a usage of EmployeesView without using EmpDeptFkLink, as in the data model shown here:

This module provides usages of the same two view objects, but the view usages are linked. Through this application module, your client application could select rows in DepartmentsView and EmployeesView usages independently.

You could even include two usages of EmployeesView in your application module: one linked with the view link and one at the top level, as in the data model shown here.



Through this application module, a client application could select rows in EmployeesView (through EmployeesView3) and have them automatically synchronized with DepartmentsView3 and could also select them (through EmployeesView4) independently. Data models will be discussed further in Chapter 13.

Client access to view objects always goes through the application module. Clients send a request to an application module to see a particular view usage, and the application module finds the view usage in its data model and passes it back to the client. Accessing application modules from clients will be discussed further in Chapter 14.

Application modules also handle transactions: starting a new one, setting locking options, and performing database posts, rollbacks, and commits. Transactions will be discussed further in Chapter 15.

Like an entity object or a view object, an application module has two parts: a Java class (such as MasterDetailModuleImpl.java) and an XML file (such as MasterDetailModule.xml). The Java class handles transaction logic and provides service methods clients can call to execute multiple operations in a batch. The XML file stores the data model.

Domains

A *domain* is a special Java type used by BC4J as the types for many entity object and view object attributes. When entity objects were introduced earlier, you learned that the class that implements the Departments entity object has some entity attributes of type oracle.jbo.domain.Number. The entity attributes of an entity object Java class are objects, not primitive Java types such as int.

For database columns of SQL datatype VARCHAR2, there is an obvious Java class for the entity attributes—java.lang.String. For other SQL types (such as NUMBER or BLOB), Business Components for Java provides domains to wrap the SQL datatype. Some domains, like oracle.jbo.domain.Number, are basically object wrappers for scalar types. Others, like oracle.jbo.domain.BlobDomain, are more complicated classes that store extensive data.

JDeveloper will also automatically create a domain for you if you base an entity object on a table with an Oracle object type column in it. This domain represents the Oracle object type, giving you Java wrappers for each of the object type's fields and methods. Domains that represent Oracle object types will be discussed further in Chapter 10.

Finally, you may create your own domains. Suppose you have many database columns, possibly in different tables, that are all very similar. Not only are they of the same SQL type (for instance, VARCHAR2), but they all contain information in exactly the same form—for example, a URL.

You may have business logic that is simultaneously associated with multiple columns. For example, you may have validation logic that applies to any and all URLs—perhaps they must all begin with a protocol code, a colon, and two slashes. Rather than putting this logic in every entity object that contains one of these columns, you can create a URL domain that itself contains the validation code. Then you simply need to ensure that the appropriate entity attributes in the entity objects' Java classes are all instances of that domain rather than the type java.lang.String. Domains that enforce business logic will be discussed further in Chapter 12. Domain code is stored in Java and XML files.

Business Components, Java, and XML

As mentioned, entity objects, view objects, and application modules each have two parts: a Java class and an XML file. The Java class and the XML file have different purposes. BC4J is a framework, which means that much of its functionality is contained in a set of libraries. Business components Java classes extend (subclass) the base classes provided in those libraries to provide complex business logic, which requires the procedural power of Java to implement. If you need that sort of logic, you can easily write code to implement complex business rules inside your entity object Java classes, complex query logic in your view object classes, or complex transaction handling in your application module classes.

However, the base classes that make up the BC4J framework are preprogrammed to work with XML files. Some business logic is common and simple and can be handled with a line or two of declarative XML. Instead of writing a procedure to implement this sort of business logic, you can just declare that an entity attribute, for example, must obey a particular validation rule. The base entity object class (and by inheritance your custom Java class) can automatically read the XML file to enforce the rule. JDeveloper provides wizards that write and edit the XML files for you.

In addition, the BC4J framework, like many other frameworks, uses XML files to store static definitions such as the structure of a database table or a query. The base classes can discover the structure of particular business components by parsing the XML. This can have performance advantages. For example, an entity object does not need to query the database at runtime for structural information about the table it represents.

Hands-on Practice: Examine a Default BC4J Layer

In this practice, you will create and explore a default BC4J layer based on the HR schema. A *default BC4J layer* is one created entirely using the Business Components Package Wizard. It is basically a prototype. For enterprise applications, you should start with some wizard-generated components but design most of your business components by hand. In addition, a default BC4J layer does not contain any business logic. However, it provides examples of business components discussed in this chapter: entity objects, associations, view objects, view links, and an application module.

This practice steps you through the following phases:

I. Create a BC4J layer

- Create a workspace and empty project
- Create default business components for the HR Schema

II. Explore a default entity object and a default association

- Explore Departments
- Explore DepartmentsImpl.java
- Explore Departments.xml
- Explore EmpDeptFkAssoc

III. Explore a default view object and a default view link

- Explore DepartmentsView
- Explore DepartmentsViewImpl.java
- Explore DepartmentsView.xml
- Explore EmpDeptFkLink

IV. Explore the default application module

- Explore Defaultbc4jModule
- Explore Defaultbc4jModuleImpl.java
- Explore Defaultbc4jModule.xml

V. Test the default Business Components

- Open the Business Component Browser
- Test an independent usage of DepartmentsView
- Test two different usages of EmployeesView
- Test EmpDeptFkLink

I. Create a BC4J Layer

This phase creates a default BC4J project.

Create a Workspace and Empty Project

A new workspace and project for your business components can be created using the following steps:

1. On the Workspaces node in the Navigator, select New Workspace from the right-click menu. The New Workspace dialog opens.
2. Use "DefaultBusinessComponentsWS" as the name for both your workspace directory and your workspace file.
3. Click OK. The New Project dialog opens.
4. Use "DefaultBC4J" as the name for both your project directory and your project file.

Create Default Business Components for the HR Schema

Default business components can be created using the following steps:

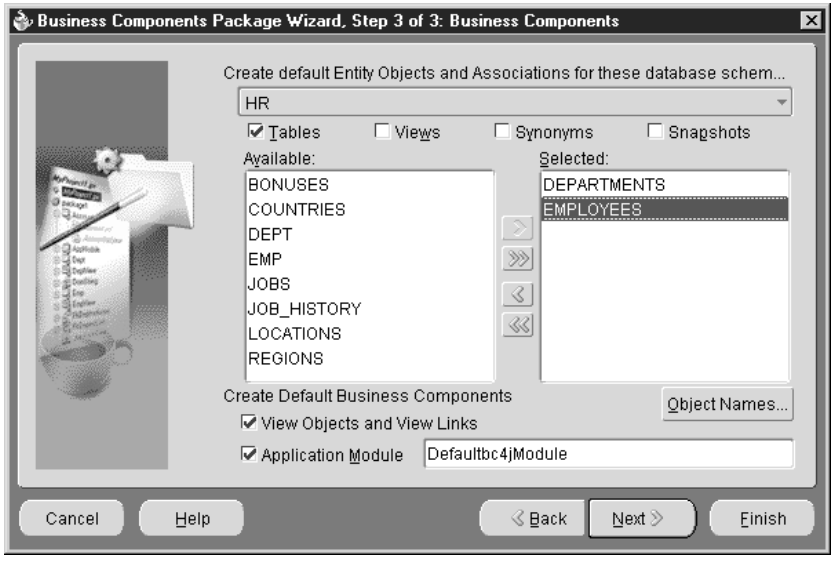
1. On the DefaultBC4J.jpr project node in the Navigator, select New Business Components Package from the right-click menu. The Business Components Package Wizard opens.
2. If the Welcome page appears, click Next.
3. On the Name page, enter "defaultbc4j" as the package name. All of your business components will go into this package.
4. Ensure that "Entity Objects mapped to database schema objects" is selected, and click Next.

   Additional Information: The other two options allow you to create EJB Entity Facades and related business components. For more information, see the sidebar "EJB Entity Facades" earlier in this chapter.
5. On the Connection page, select HR from the *Connection Name* dropdown. Click Next.
6. On the Business Components page, use the right arrow to add DEPARTMENTS and EMPLOYEES to the *Selected* list. This will create an entity object from each of those tables and create associations from each of the foreign keys between them.
7. For this practice, leave the *View Objects and View Links* checkbox checked.

   Additional Information: This will create a default view object for each entity object and one view link for each association. As you will see later, default view objects and view links are not very useful for many real applications. (They correspond to SELECT * single-table queries and foreign-key–based master-detail relationships rather than the more complex queries and relationships that most applications need.) But they are useful for testing the functionality of the entity objects.
8. Also leave the *Application Module* checkbox checked, with the name "Defaultbc4jModule" filled in.

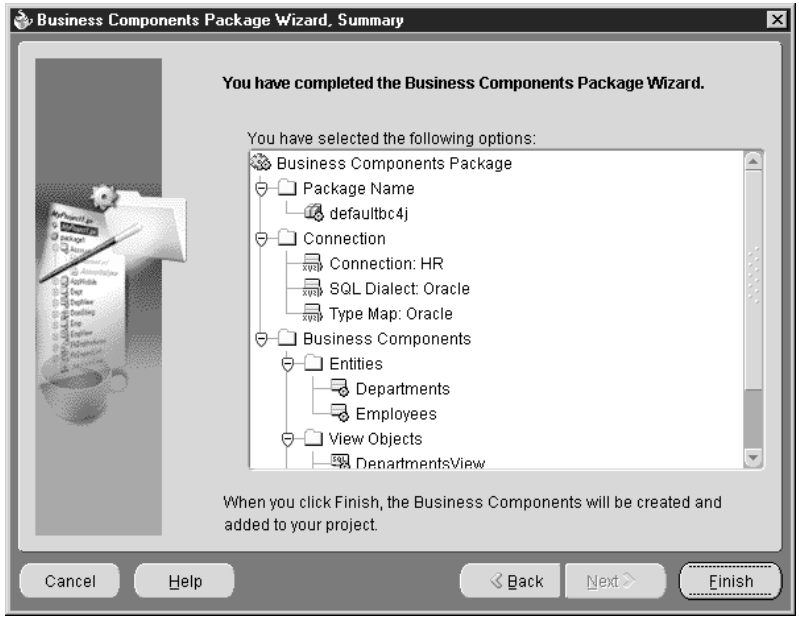
   Additional Information: This will create a default application module that contains the view objects in every possible combination, both independent and joined by view links. Again, the default application module would not be that useful for developing

a real application (it is generally too big and inefficient), but it is useful for testing a business components project. The page should look like the following illustration:



9. Click Next. The Summary page appears.

Additional Information: The Summary page lists connection information, the package name, and the primary business components (entity objects, view objects, and the application module) that the wizard will create, as shown next:



10. Click Finish.
11. Select **File | Save All**.

What Just Happened? You created a BC4J layer. JDeveloper creates business components based on the choices you made in the wizard. The business components are under the defaultbc4j node in the Navigator, as shown in Figure 9-1.

II. Explore a Default Entity Object and a Default Association

This phase looks at the Departments entity object, which the wizard created based on the DEPARTMENTS table, and EmpDeptFkAssoc, the association which the wizard created based on the EmpDeptFk constraint from the database.

Explore Departments

You can view an entity object's entity attributes and their types using the Entity Object Wizard. This is a low-level wizard you would use to create or edit a non-default entity object.

1. In the System Navigator, expand the defaultbc4j node. On the Departments node, select Edit Departments from the right-click menu. A re-entrant version of the Entity Object Wizard appears.

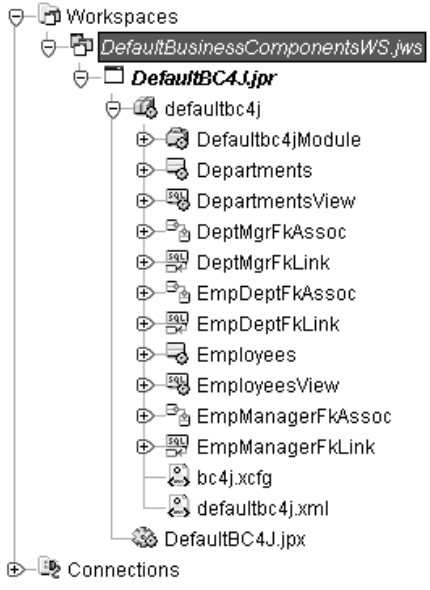
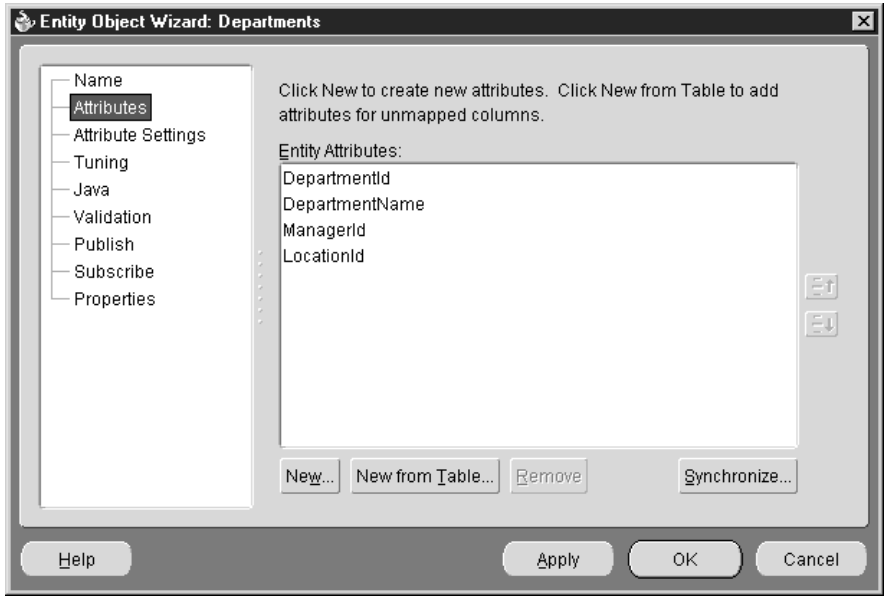


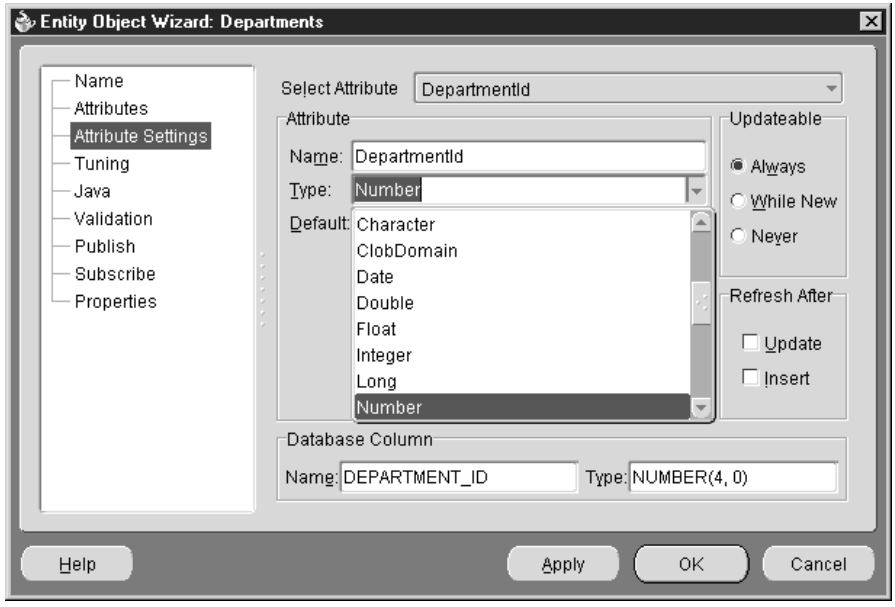
FIGURE 9-1. *The default business components*

2. Select the Attributes node. You will see a list of the entity object's attributes, containing one attribute for each column in the DEPARTMENTS table, as shown here:



3. Select the Attribute Settings node. The first attribute, DepartmentId, will be listed in the *Select Attribute* field. Its Java type is Number.

4. Open the *Type* dropdown list to see the options for Java types of attributes, as shown here:



Additional Information: Of the types visible in the preceding illustration, ClobDomain, Date, and Number are all domains; the others are classes from the package java.lang.

5. Click Cancel to close the wizard without making any changes.

Explore DepartmentsImpl.java

Examine an entity object's Java source:

1. In the System Navigator, expand the Departments node. You will see the two entity object files: Departments.xml and DepartmentsImpl.java.
2. Double click DepartmentsImpl.java to open its source code.
3. Look in the Structure window for methods that start with "get" and "set." These are the methods that retrieve and change the values of each attribute.

   Additional Information: The Java class associated with each entity object has accessor methods that retrieve and change the values of each attribute. For example, the DepartmentsImpl class has methods `getManagerId()` and `setManagerId()` that retrieve and change the values of the ManagerId attribute. When an entity object Java class is instantiated, the resulting Java object represents one row of the database table. For example, if departmentsImpl1 is a particular DepartmentsImpl object, its getters would return values matching the values in a row of the DEPARTMENTS table as shown here:

Method Called	Value Returned
departmentsImpl1.getDepartmentId()	A Number domain holding the value 10
departmentsImpl1.getDepartmentName()	The String "Administration"
departmentsImpl1.getManagerId()	A Number holding the value 200
departmentsImpl1.getLocationId()	A Number holding the value 1700

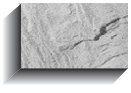
In Chapter 12, you will learn how to change these methods to enforce business rules.

Explore Departments.xml

Examine an entity object's XML metadata:

1. Double click Departments.xml to open its source.
2. Find the <Attribute> tags. There is one for each attribute: DepartmentId, DepartmentName, ManagerId, and LocationId.

   Additional Information: The <Attribute> tags store metadata about the individual attributes, including the Java type of the attribute, the identity and SQL datatype of the column it maps to, and simple validation added with the Entity Object Wizard.



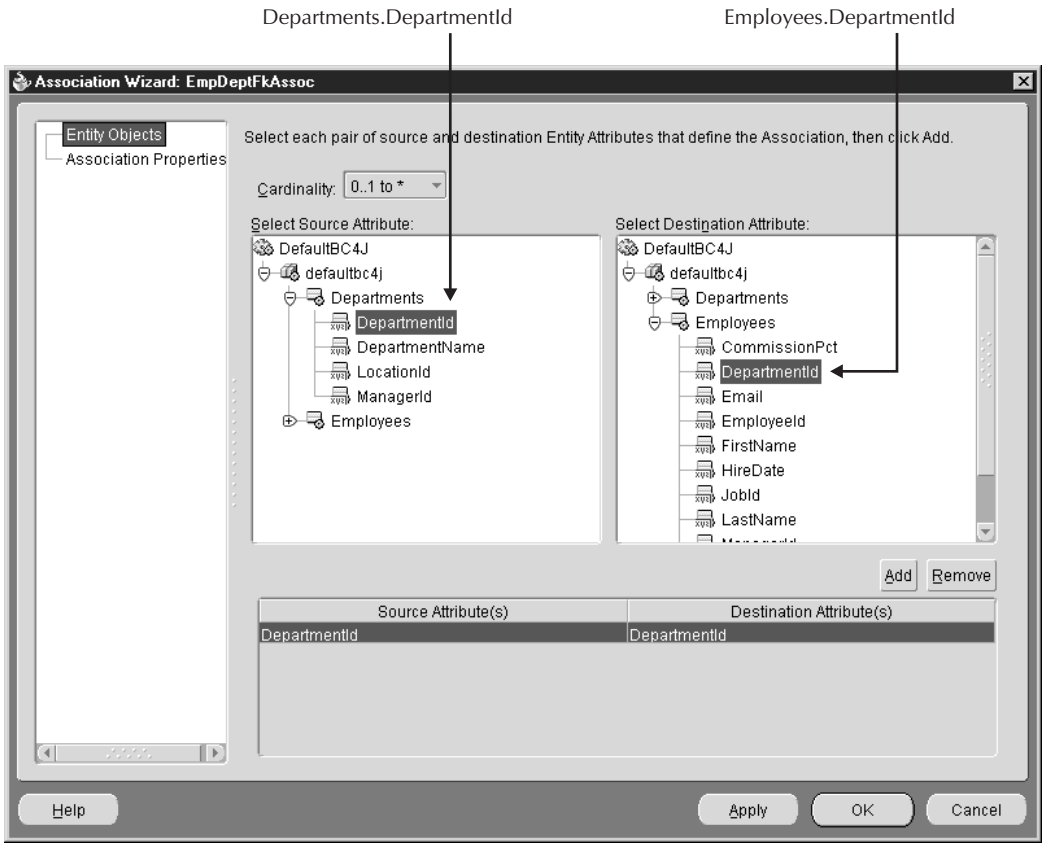
TIP

When an XML file is loaded into the Code Editor, you can expand the nodes of the Structure window to examine a summary of the structure of the file.

Explore EmpDeptFkAssoc

You can examine an association using the Association Wizard, a low-level wizard which you would use to edit or create a non-default association:

1. On the EmpDeptFkAssoc node in the Navigator, select Edit EmpDeptFkAssoc from the right-click menu. A re-entrant version of the Association Wizard appears.
2. Note that the source entity object (which contains the primary key) is Departments, and the destination entity object (which contains the foreign key) is Employees. The wizard page shows the primary key attribute under the source entity object node and the foreign key attribute under the destination entity object node.



3. Click Cancel to close the wizard without making any changes.

What Just Happened? You looked at an entity object and an association. The entity object's Java file is where you would place procedures to enforce business rules; the XML file contains the

metadata (such as datatypes) and declarative rules created using the Entity Object Wizard. The association links the entity object with another entity object, just as foreign key relationships link two tables.

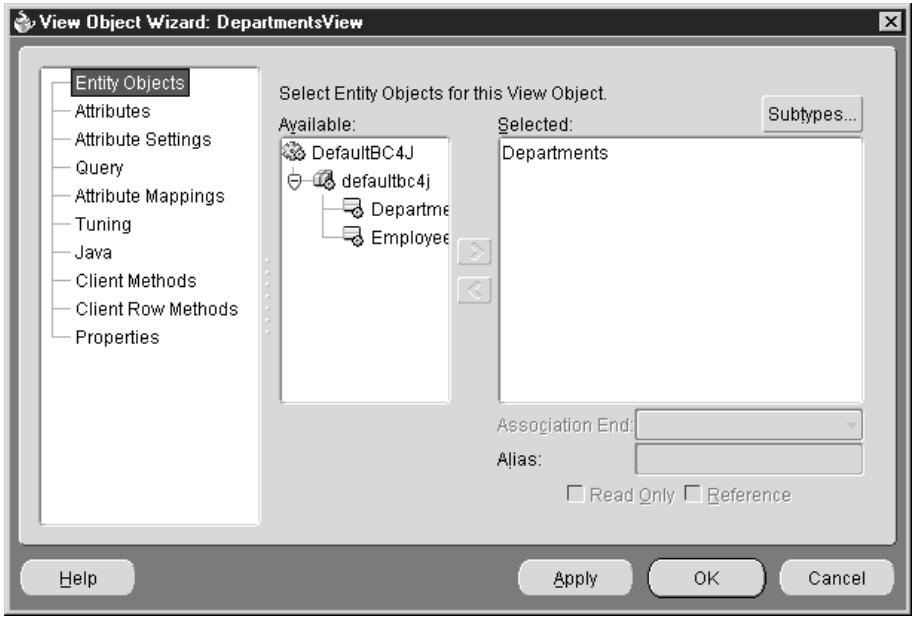
III. Explore a Default View Object and a Default View Link

This phase examines DepartmentsView, the view object that the wizard created based on the Departments entity object, and EmpDeptFkLink, the view link that the wizard created between DepartmentsView and EmployeesView, another default view object.

Explore DepartmentsView

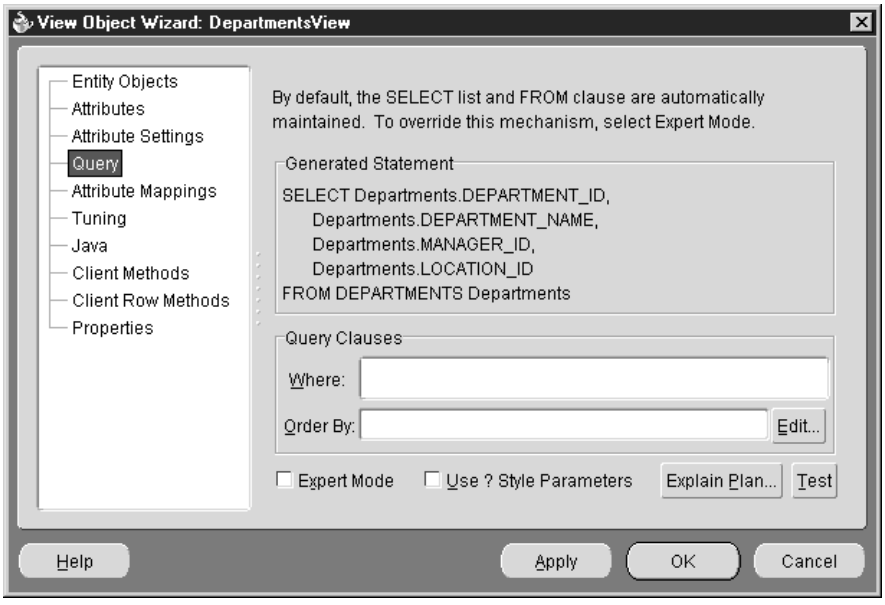
You can examine a view object's query and the entity objects on which it is based by using the View Object Wizard:

1. On the DepartmentsView node in the Navigator, select Edit DepartmentsView from the right-click menu. A re-entrant version of the View Object Wizard appears.
2. Note that this view object is based on the entity object Departments, as shown here in the *Selected* pane in the following illustration:



Additional Information: By using the arrow buttons, you could change this, and base DepartmentsView on either entity object, both, or neither.

3. Select the Query node to see the view object's query, as shown next.



Additional Information: Default view objects such as DepartmentsView query every column in every row of the database table. When you create your own view objects, you probably will not want to do this. Instead, you will tailor your query to the application's data needs.

4. Click Cancel to close the View Object Wizard without making any changes.

Explore DepartmentsViewImpl.java

A view object's Java source can be examined using the following steps:

1. In the System Navigator, expand the DepartmentsView node. You will be able to see the two view object files: DepartmentsView.xml and DepartmentsViewImpl.java.
2. Double click DepartmentsViewImpl.java to open its source code.
3. Look in the Structure window or directly at the code and note that DepartmentsViewImpl extends ViewObjectImpl, but has no accessor methods.

   Additional Information: Unlike the Java class for an entity object, a view object class does not have getters and setters for its attributes. An instantiation of a view object class does not correspond to a row, but instead to the entire results of the query. That instantiation has several methods, such as `first()`, `next()`, and `findByKey()`, inherited from ViewObjectImpl, which return objects called *view rows*. These do correspond to rows of returned data in a query, and they do have methods, `getAttribute()` and `setAttribute()`, which return and set values for the view attributes. For example, if DepartmentsViewImpl1 is a particular

DepartmentsViewImpl usage, `DepartmentsViewImpl1.first()` would return a view row with the ability to return the values of a row of the query as in the following table:

Method Called	Value Returned
`DepartmentsViewImpl1.first().getAttribute`("DepartmentId")	A Number holding the value 10
`DepartmentsViewImpl1.first()..getAttribute`("DepartmentName")	"Administration"
`DepartmentsViewImpl1.first().getAttribute`("ManagerId")	A Number holding the value 200
`DepartmentsViewImpl1.first().getAttribute`("LocationId")	A Number holding the value 1700

You can use the DepartmentsViewImpl class to override the methods inherited from ViewObjectImpl or to add your own methods to implement custom query logic.

Explore DepartmentsView.xml

A view object's XML file can be examined using the following steps:

1. Double click DepartmentsView.xml to open its source.
2. Find the <ViewObject> tag close to the top of the file.

   Additional Information: The top-level element of the XML file, <ViewObject>, has XML attributes that together specify the query. The SelectList XML attribute specifies the columns in the query's SELECT clause and the FromList XML attribute lists the tables in its FROM clause. If the query had a WHERE clause, it would be specified with a Where XML attribute.
3. Find the <EntityUsage> tag for the Departments entity.

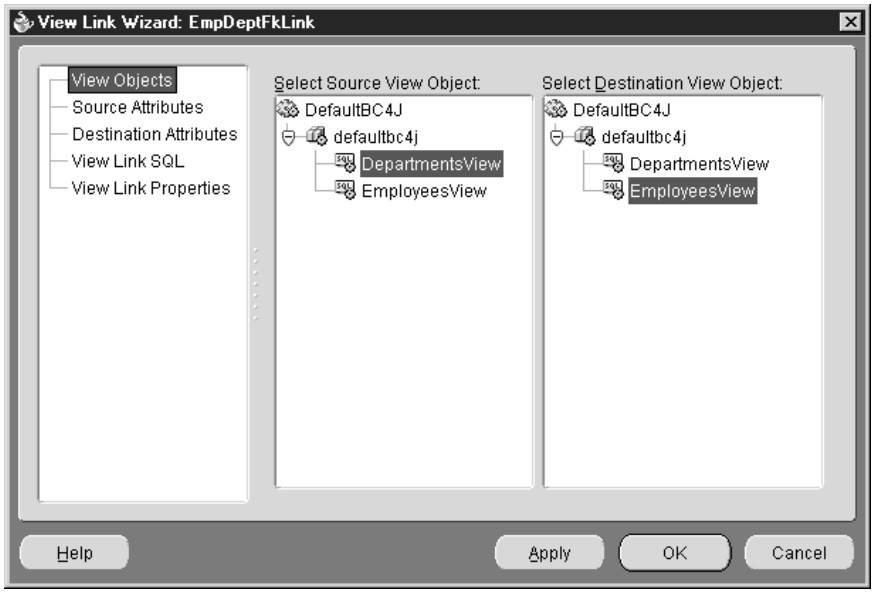
   Additional Information: A view object's attributes can be, but do not need to be, associated with entity attributes. If at least some view attributes are associated with entity attributes, the view object's XML file will contain elements called <EntityUsage> that relate the view object to the entity objects.
4. Find the <ViewAttribute> tags.

   Additional Information: A view object's XML file contains a <ViewAttribute> element for every view attribute. These elements look a bit different depending upon whether the view attribute is associated with an entity attribute. Since all of the view attributes in DepartmentsView are based on entity attributes in Departments, all of its <ViewAttribute> tags contain references to the Departments entity and its attributes.

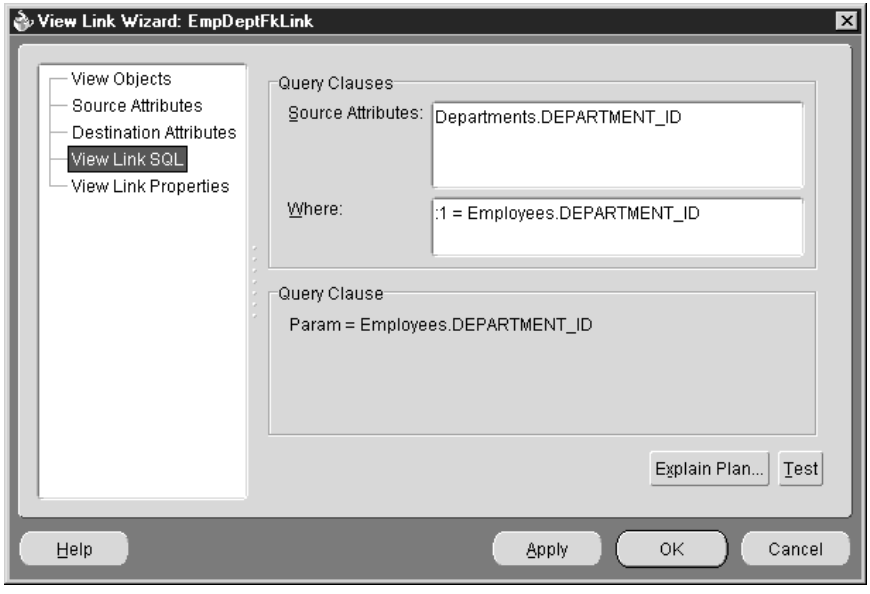
Explore EmpDeptFkLink

You can examine a view link using the View Link Wizard:

1. On the EmpDeptFkLink node in the Navigator, select Edit EmpDeptFkLink from the right-click menu. A re-entrant version of the View Link Wizard appears.
2. Note that the view link joins the DepartmentsView view object and the EmployeesView view object, as shown next.



3. Click the View Link SQL node. In the Query Clauses box, you can see the WHERE clause that the view link will use to join Departments data to Employees data, shown next:



Additional Information: The parameter DepartmentsId from a row of the source view object will be substituted for :1 in the WHERE clause "WHERE :1=Employees.DEPARTMENT_ID" to limit the rows returned by a detail usage

of EmployeesView. This definition is the reason why you will only be able to see employees for the selected department if you use this view link in your data model.

4. Click Cancel to close the wizard without making any changes.

What Just Happened? You looked at a view object and a view link. The view object's Java file is where you would place procedural logic that requires traversing the rows of the query's result. The XML file contains the metadata and declarative rules you create using the View Object Wizard. The view link links another view object to this one using a WHERE clause.

IV. Explore the Default Application Module

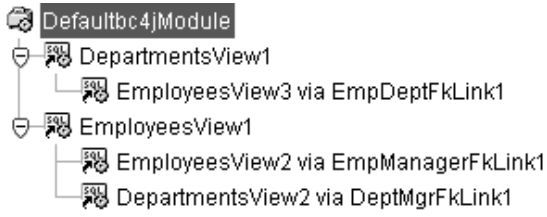
This phase looks at Defaultbc4jModule, the default application module created by the wizard.

Explore Defaultbc4jModule

You can examine an application module's data model using the Application Module Wizard:

1. On the Defaultbc4jModule node in the Navigator, select Edit Defaultbc4jModule from the right-click menu. A re-entrant version of the Application Module Wizard appears.
2. Examine the application module's data model.

   Additional Information: The data model of a default application module contains all possible view objects in all possible combinations: both independent and joined by view links. When you create your own application modules, you probably will not want to do this. Instead, you will use just the view objects your application needs, in just the combinations it needs. The following is the data model for Defaultbc4jModule:



   Additional Information: DepartmentsView1 and EmployeesView1 are independent usages of the DepartmentsView and EmployeesView view objects, respectively. The other usages are linked to those usages as details in master-detail relationships.

3. Click Cancel to close the wizard without making any changes.

Explore Defaultbc4jModuleImpl.java

An application module's Java source can be examined using the following steps:

1. In the System Navigator, expand the Defaultbc4jModule node to see the two application module files: Defaultbc4jModule.xml and Defaultbc4jModuleImpl.java.
2. Double-click Defaultbc4jModuleImpl.java to open its source code.

3. Use the Structure window to find the following methods, and double-click each to jump to the code for the method:

```
getDepartmentsView1()
getEmployeesView1()
getEmployeesView2()
getDepartmentsView2()
getEmployeesView3()
```

Additional Information: These methods return view object usages in the application module. You can call them from your client applications to access particular view object usages.

Explore Defaultbc4jModule.xml

An application module's XML file can be examined using the following steps:

1. Double click Defaultbc4jModule.xml to open its source.
2. Find the <ViewUsage> elements for each view object usage in the application module.
3. Find the <ViewLinkUsage> elements for each of the three times view links were used in the data model.

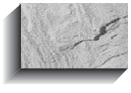
   Additional Information: An application module's XML file contains the element <ViewLinkUsage> for each time a view link was used in creating the data model. The attributes SrcViewUsageName and DstViewUsageName specify the view usages that govern and are governed by the view link, respectively. A DstViewUsageName value of "defaultbc4j.Defaultbc4jModule.EmployeesView2" indicates that the detail view usage governed by the view link is EmployeesView2.

What Just Happened? You looked at an application module. The application module's Java file is where procedural transaction logic can be implemented and where you can find methods that return the view object usages. The XML file contains the metadata and declarative rules that specify your data model.

V. Test the Default Business Components

Now that you have examined the business components, you can look at them in action. Business components do not have a user interface, so it is not really possible to see the business components working except through a user interface.

JDeveloper includes a user interface, the Oracle Business Component Browser, that can be run to test any application module. The browser will be used to test Defaultbc4jModule. Because Defaultbc4jModule contains all possible view objects in all possible combinations, testing Defaultbc4jModule is really a way of testing the entire business components project.



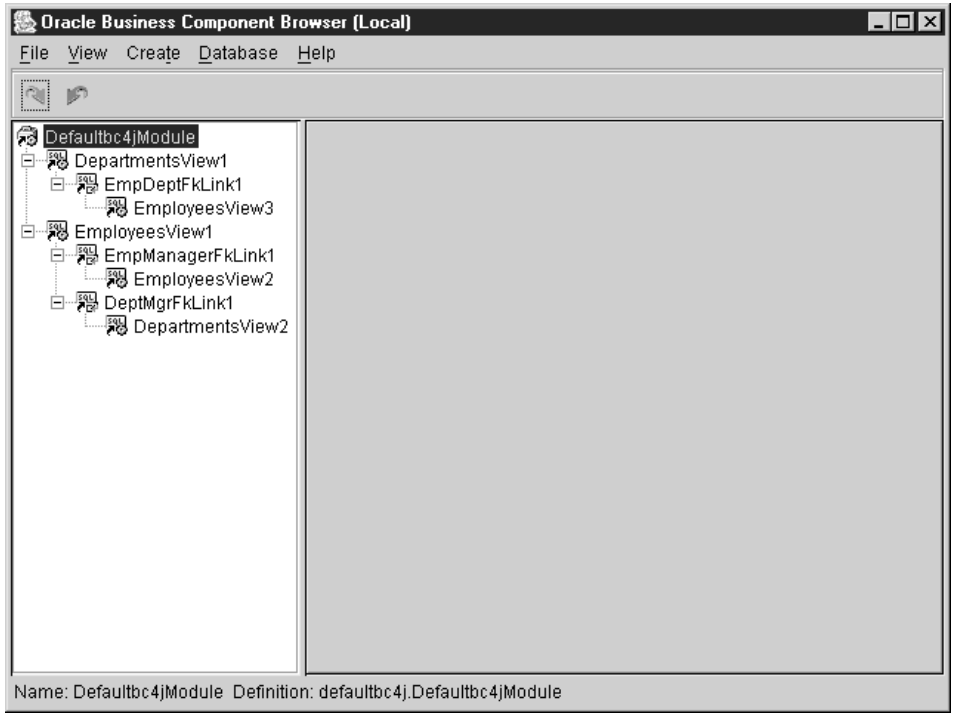
NOTE

As mentioned earlier, default application modules are not generally good for production-quality enterprise applications because they are too big. They contain usages of every view object, linked together in every possible combination, instead of just the data model the client needs. This leads to a needlessly large cache, which can degrade performance. However, they are excellent for testing all of your business components at once.

Open the Business Component Browser

The Business Component Browser can be opened to test an application module and its associated business components using the following steps:

1. On the Defaultbc4jModule node, select Test from the right-click menu. The Business Component Browser starts up, showing the Connect dialog.
2. The defaults in the Connect dialog are fine for the purposes of this practice, so click Connect. The main dialog of the Business Component Browser appears. In the left-hand pane, you can see the Defaultbc4jModule's data model, as shown next:



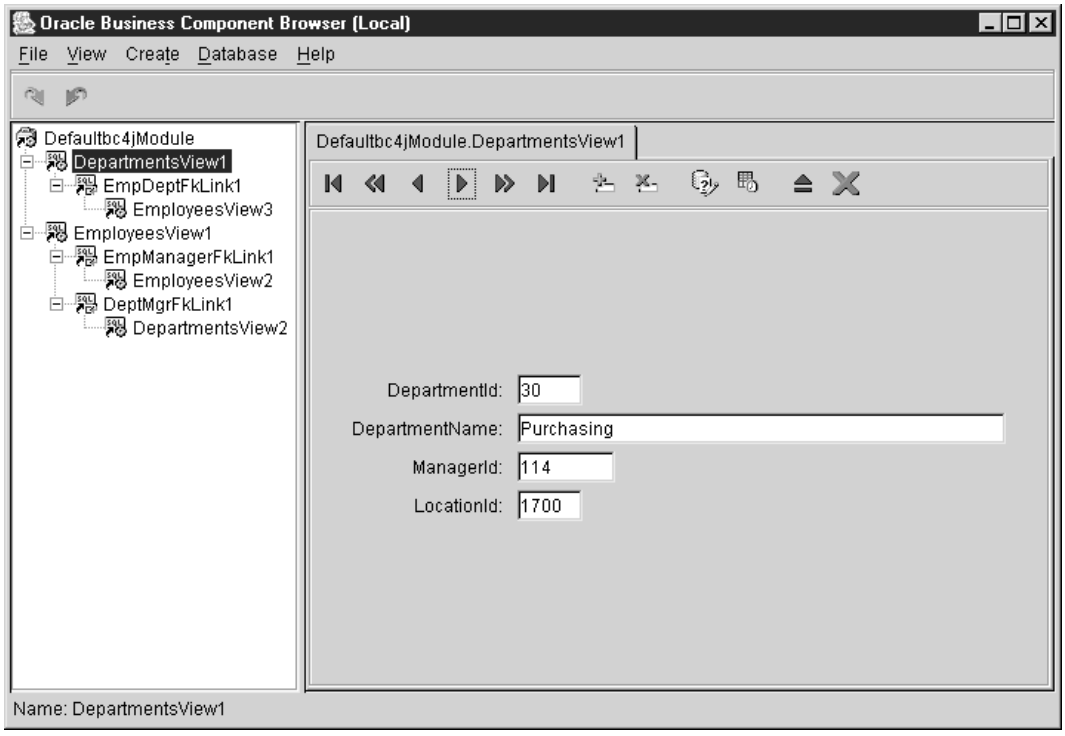
TIP

You can also click Run in the IDE toolbar after selecting the application module or BC4J project node in the Navigator. This method bypasses the Connect dialog.

Test an Independent Usage of DepartmentsView

You can scroll through the query results for an independent view object usage using the following steps:

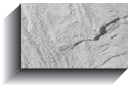
1. Double click DepartmentsView1. In the right-hand pane, you can now see the data from DepartmentsView's query, one row at a time.
2. Click the blue right arrow to scroll forward to Department 30 as shown next:



Test Two Different Usages of EmployeesView

You can contrast detail and independent usages of a view object using the following steps:

1. Be sure that department 30 is displayed. Double click EmployeesView3 in the left-hand pane. In the right-hand pane, you can now see data from EmployeesView's query, one row at a time.
2. Scroll forward through the data. Notice that all the employees listed have DepartmentId 30. This is because the business components framework uses EmpDeptFkLink to automatically synchronize EmployeesView2 with DepartmentsView. If the current row of DepartmentsView is 30, only employees with that department will show up in EmployeesView3.
3. Double click EmployeesView1. The same kind of single-row browser is displayed.
4. Scroll forward through the data. Notice that EmployeesView1 contains all of the employees, not just those with DepartmentId 30. That is because the EmployeesView1 usage, unlike the EmployeesView3 usage, is not governed by DepartmentsView1.
5. Click the red X in the browser toolbar to close each open window in turn.



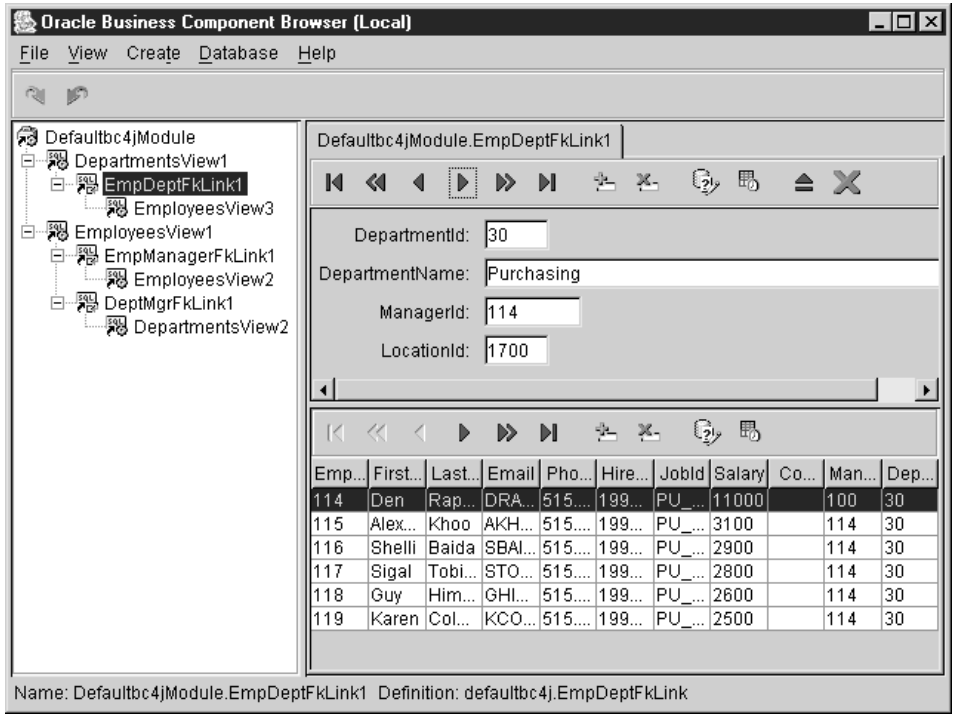
TIP

If you want to pull a window out of the tester's frame, click the blue up arrow in the browser toolbar.

Test EmpDeptFkLink

You can view a split-screen display of joined view objects using the following steps:

1. Double click EmpDeptFkLink1. The right-hand pane is now split. In the top half, you can see rows from DepartmentsView1 one at a time, and in the bottom half, you can see a table of all rows currently showing in EmployeesView3. This is simply another way the Business Component Browser lets you view linked view objects. Selecting a view link lets you view both objects at once, as shown next:



2. Scroll through the departments. The list of employees automatically changes.
3. Close the Business Component Browser.

What Just Happened? You used the Business Component Browser to test your business components. You saw a display of all the view usages in your data model, two ways of displaying data in the Business Component Browser, and how view links synchronize detail view usages with their master view usage.